CSDL-T-1158

AN OBJECT-ORIENTED DYNAMIC SOFTWARE PROCESS MODEL

by

Bradley J. Smith

January 1993

Master of Science Thesis
Boston University

DRAPER LABORATORY

The Charles Stark Draper Laboratory, Inc.
555 Technology Square, Cambridge, Massachusetts 02139-3563

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
Jan 1993

3. REPORT TYPE AND DATES COVERED
THESIS/~~DISSERTATION~~

4. TITLE AND SUBTITLE
An Object-Oriented Dynamic Software Process Model

5. FUNDING NUMBERS

6. AUTHOR(S)
Capt Bradley J. Smith

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student Attending: Boston University

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA- 93-092

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPARTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distribution Unlimited
MICHAEL M. BRICKER, SMSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
99

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

AN OBJECT-ORIENTED DYNAMIC SOFTWARE PROCESS MODEL

by

Captain Bradley Jay Smith, USAF
Master of Science, Computer Engineering
Boston University, College of Engineering, 1993
Length: 100 Pages

Abstract

This thesis describes a new method for modeling complex dynamic processes using object-oriented techniques. These techniques are applied to develop a sophisticated model of the software development process. This object-oriented approach has many advantages over non-object-oriented System Dynamics models. Non-homogeneous resource allocation can be explicitly modeled. This allows resource allocation strategies and management decisions to be modeled and assessed in a realistic manner. Since software development tasks and resources are modeled as objects, they can be flexibly created and manipulated to capture the dynamics of a particular project. Furthermore, the principles of object-oriented inheritance lets the modeler or analyst easily extend the model to include new resources and tasks. The thesis addresses major considerations in modeling the software development process, as well as the detailed development of an object-oriented software process model in C++. A comparison is done between this dynamic model and the empirical predictions of the popular COCOMO software estimation system. This new object-oriented approach is not limited to software process modeling and could be applied to a wide variety of scientific, business, and social simulations.

Major References

1. Tom DeMarco and Timothy Lister, Peopleware: Productive Projects and Teams, Corset House Publishing Co. , New York, 1987, p.188
2. Barry Boehm, Software Engineering Economics, The COCOMO Software Model, 1980
3. Roger Pressman, Software Engineering, A Practicioner's Approach,
4. McGraw-Hill, 1992 p. 87
5. High Performance Systems Inc, Stella II User's Guide, 1990
6. Tarek Abdel-Hamid and Stuart Madnick, Software Project Dynamics, 1991
7. Smith, A. Clough, R. Vidale, N. Nguyen, S. Ahmed, The Software Process Model, Draper Labs, May 1992
8. Bradley Smith, Proposal for Independent Study of an Object Oriented Software Process Model, May 1992
9. Grady and Caswell, Software Metrics Prentice Hall, 1987, p. 24-25
10. Borland International, Borland C++ Programmer's Guide, Version 3.1, 1992
11. Borland International, ObjectWindows for C++ User's Guide, Version 3.1, 1992
12. Microsoft Inc., Microsoft Windows User's Guide, Version 3.1, 1992

BOSTON UNIVERSITY

GRADUATE SCHOOL

Thesis

AN OBJECT-ORIENTED DYNAMIC SOFTWARE PROCESS MODEL

by

Bradley Jay Smith

B.S., Rensselaer Polytechnic Institute, 1986
M.B.A., St. Mary's University, 1989

Submitted in partial fulfillment of the

requirements for the degree of

Masters of Science, Computer Engineering

1993

Approved by

First Reader: Richard F. Vidale, Ph.D.
Professor of Electrical, Computer and Systems Engineering

Second Reader: Thomas D. C. Little, Ph.D.
Assistant Professor of Electrical, Computer and Systems Engineering

Acknowledgements

12/10/92

This thesis was prepared at The Charles Stark Draper Laboratory Inc., under Internal Research and Development Contract #349.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

Dedication

To my wife and daughter.

Many thanks to the people of Draper Labs including, but certainly not limited to:

N. Nguyen, A. Clough, S. Ahmed

Thanks also to the Boston University Advisors:

Prof. Richard Vidale and Prof. Thomas Little

AN OBJECT-ORIENTED DYNAMIC

SOFTWARE PROCESS MODEL

(Order No.)

by

Bradley Jay Smith

Boston University, College of Engineering, 1992

Major Professor: Richard F. Vidale
Professor of Electrical, Computer and Systems Engineering

Abstract

This thesis describes a new method for modeling complex dynamic processes using object-oriented techniques. These techniques are applied to develop a sophisticated model of the software development process. This object-oriented approach has many advantages over non-object-oriented System Dynamics models. Non-homogeneous resource allocation can be explicitly modeled. This allows resource allocation strategies and management decisions to be modeled and assessed in a realistic manner. Since software development tasks and resources are modeled as objects, they can be flexibly created and manipulated to capture the dynamics of a particular project. Furthermore, the principles of object-oriented inheritance lets the modeler or analyst easily extend the model to include new resources and tasks. The thesis addresses major considerations in modeling the software development process, as well as the detailed development of an object-oriented software process model in C++. A comparison is done between this dynamic model and the empirical predictions of the popular COCOMO software estimation system. This new object-oriented approach is not limited to software process modeling and could be applied to a wide variety of scientific, business, and social simulations.

Table of Contents

List of Tables

List of Figures

Problem Overview

The challenge of developing software on-time and within budget has grown exponentially over the past ten years, mirroring the exponential growth in size and number of software projects. Of the 500 projects Tom DeMarco has studied, a full 15% produced nothing. Of the projects of 25 or more person-years, 25% failed.[1] The complexity of managing the development of a large software project requires more efficient tools for planning and controlling the software development process.

Types of Planning Tools

- ***Empirical or Traditional Tools:*** Current project management tools make empirical estimates of time and manpower to complete a project based largely on the size or number of functions in the project. These estimates are often based on statistical studies of historical data collected from a number of projects. This group of tools includes such popular planning tools as Barry Boehm's COCOMO estimation model,[2] and the Putnam Estimation Model.[3] These models have limitations in that they do not take into account variables that may change with time including management decisions, new requirements, errors generated, and the state of the work force. The empirical modeling approach will not be discussed in great detail here. For a complete discussion see Pressman's Software Engineering book.[4] These tools are widely in use today, and most software managers have used at least one of these tools in software development.

- ***Dynamic Modeling Tools:*** The dynamic software process model differs from the empirical in that it attempts to model the behavior of the development process over time. This type of modeling is based on the System Dynamics techniques developed by Forester and others at the Sloane School of Business at MIT.[5] These models use a system of linear differential equations to determine the interactions between the people, process, and work

accomplished. The first major application of this modeling technique to the software process was done by Abdel-Hamid and Madnick.[6] They developed a software process model using the DYNAMO modeling language to model the tradeoffs in manpower, errors and work completed in a typical development process.

In 1991, the Charles Stark Draper laboratory began a research and development project in dynamic software process modeling. This Internal Research and Development produced Draper's Software Process Model.[7] The Draper Software Process Model expands on the work of Abdel-Hamid and Madnick by incorporating a number of additional factors and effects. The Draper Process model uses the same system dynamic principles as the Abdel-Hamid model, and is implemented in a graphical modeling language called STELLA.™ The Draper model will be used as the basis for comparison when evaluating the new object-oriented approach to modeling described in this thesis.

- ***Object-Oriented Dynamic Tools:*** The main purpose of this thesis is to describe a new object-oriented approach to dynamic modeling. This approach uses an object-oriented structure and programming language to implement a dynamic model of the software process. This model uses the same System Dynamics principles as the dynamic software process models described above, but the model is implemented in an object-oriented language rather than currently available modeling languages like DYNAMO™ or STELLA™. The main advantage of this approach is a large increase in flexibility when modeling other types of processes.

Limitations of the Draper Software Process Model

Draper's approach to implementing a model of the software process used a graphical modeling language tool called STELLA™ to model a waterfall software development process.[8] A fairly sophisticated system dynamics model was built using STELLA™ to model production, error creation, the state of manpower, and a number of interacting effects. This prototype model

expanded greatly on the work of Abdel-Hamid by explicitly detailing error feedback effects in each phase, incorporating additional productivity factors, modeling phases in the waterfall individually, and allowing a manager to make decisions interactively during a modeling session.

This prototype model has met with some success as a training and planning tool. One of the Draper model's major limitations, inherent in the use of current modeling tools like STELLA™, is that the process model is not particularly flexible. To apply the model to a Spiral or Incremental process for example would require rewriting the model. Even a simple task like modeling two different development teams would require major changes to the model.

Further, the original model implementation is limited in the types of resources it can easily represent. It would be very difficult to distinguish the separate skills of different groups of people. It is hard to model the differing abilities of people who do analysis, programming, design and testing, for instance. Using current modeling tools, a different manpower subsystem would need to be implemented for each group of specialist.

While the model is excellent for modeling the traditional waterfall life-cycle project, done by a homogeneous group of workers, it is very difficult to model any other type of process done by non-homogeneous workers.

Key Object-Oriented Concepts

A number of key features of object-oriented languages make an object-oriented approach appropriate for modeling. These include:

- **Data Abstraction.** Each class in an object oriented language may be used arbitrarily as a data type much like integers or character strings are used in non-object oriented languages. In modeling, this allows a modeler to define abstract types to represent modeled entities.

- **Data Hiding.** Objects may be treated as black boxes with defined interfaces. The internal data members and implementation can be hidden from the user. This allows a modeler to use

an object without worrying about internal implementation details. This significantly eases the use and reuse of objects.

- **Inheritance.** Inheritance is a property that allows objects to be built from other objects. The advantage is that a modeler can inherit all of the properties of an object, and then either modify or add to its properties and behavior. This allows a modeler to build a hierarchy of objects from simple to more complex without re-implementing common features. A modeler can eventually develop a library of common modeling objects that can be used or reused in new models.

- **Messages.** Objects communicate via messages. A message is essentially a command that one object sends to another. A message is an abstract command. This process is conceptually similar to a function call, but objects need not respond in the same way to the same message. For example, the user might send a message to tell some modeling objects to step forward one time step. The objects might respond to this message in different ways depending on what is necessary for them to step forward one time step.

- **Methods and Member Functions.** An object contains both an abstract collection of data and the methods that the object performs. These methods are implemented as member functions for the object. Since different objects may have different member functions associated with them, they may behave differently. Using inheritance we can make two objects behave in different ways to the same message. For example, a *Task* object and a *Resource* object in the model can both process a *step()* message, but the two objects will have vastly different behaviors when they receive that command.

Anticipated Advantages of OOPS Method

To overcome the limitations of current modeling tools outlined above, a prototype of a software process model in the C++ object-oriented language was developed. Using an object-

oriented language allows greater flexibility in adapting the model to different development processes.[9] Specifically:

- Process objects can be defined and assembled in an arbitrary order, letting one model the traditional waterfall life cycle model as well as many other software development paradigms.
- Process objects can be derived from other process objects using inheritance. This allows us to define new process elements with minimal work.
- Resource objects can be defined generically, allowing us to model manpower as well as other development assets. New resources can be defined by inheritance directly from these base resource objects.
- Groups of resources with distinct capabilities can be modeled. In particular groups of manpower that have different capabilities (i.e. analysts, programmers, testers) can be modeled with different productivity factors for each task.
- Resource allocation strategies can be modeled and evaluated. Groups of resources can be assigned to tasks in a multitude of ways. With the object-oriented model one can explore different allocation strategies and their impact.

Process Features

To model the software process, extensive research was conducted to determine the essential characteristics of the process. The research done for the original Draper Software Process Model[10] as well as Abdel-Hamid's[11] model were drawn heavily upon to develop the object-oriented model.

The Waterfall Software Process

The starting point for the design of the object-oriented model was the traditional waterfall process as specified in Mil-Standard 2167A.[12] From the standard waterfall process we selected a simplified waterfall process to use as a basis for the model. The simplified waterfall process with error feedback is shown in Figure 1.

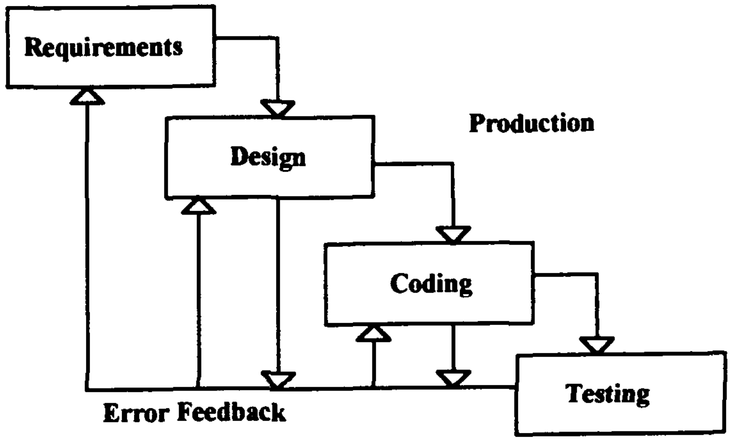


Figure 1
Waterfall Development Process

The waterfall diagram represents a portion of a traditional software development life cycle. Production flows from requirements, to design, coding and testing. Errors discovered in later phases are returned to earlier phases to be reworked. The selection of phases was made largely to facilitate calibration of the model. However, using object-oriented techniques, new tasks may be dynamically added to the model to duplicate other development paradigms.

Productivity

Normal production is modeled in the task object. Productivity is determined by the nominal productivity of the individual resources applied to a project. The nominal work rates for a project were determined initially by research from a baseline of 10 lines/day per person based on the reference *Software Metrics*.[13] This rate was subsequently adjusted during model calibration to

reflect COCOMO productivity rates and modeling effects. The final calibrated rate used was 13 lines/day.

Each resource object has productivity objects associated with it for different tasks. In the general model, each resource is assigned a productivity rate based on the overall nominal rate described above. It is possible to assign different resources unique productivity's to reflect individual talents. Also productivity of a resource may be affected by other factors such as overtime and schedule pressure. These are discussed further in the section on *Resource Attributes*.

Productivity by Phase

The nominal productivity in the sample model is divided among the different tasks in the waterfall to determine individual task productivity's. The baseline figure of 13 lines/day represents overall productivity for the process. This overall productivity was divided out for the four tasks in the waterfall process. The following table illustrates the time breakout for a typical project. Grady-Caswell[14] estimate Requirements: 14.8%; Design: 20.1%; Coding: 37.4%; and Documentation: 27.7%. In our model we rounded these figures to arrive at the following breakout (Table 1)

Table 1
Percent of Time in each Phase

Phase	% of Time
Requirements	15%
Design	20%
Coding	40%
Testing	25%

Process Rigor

A unique feature borrowed from the Draper Software Process Model[15] is the notion of rigor in a development process. The idea behind rigor is that there is a subjective measurement of the degree to which a manager maintains rigorous development and testing process. For

example, if a development process is done ad-hoc, it is considered non-rigorous. Conversely if a development process is first carefully planned, requirements are documented in detail, and the code faces regular reviews and verification, we consider the process rigorous. It is assumed assertion that less rigorous processes generally speeds the process but leads to more errors. Conversely a more rigorous process takes more time to complete but produces less errors.

In the object-oriented model each productivity object has a rigor variable associated with it. Managers may adjust the rigor for each resource productivity to reflect individual resource abilities. A rigor value of 1.0 is considered nominal, with 1.5 corresponding to roughly 2/3 nominal productivity and 2/3 the nominal error rate. Conversely a rigor setting of 0.5 will result in roughly double the productivity and double the error rate.

Error Modeling

A critical feature of the object-oriented software process model is the explicit tracking of errors. The following system borrows heavily from the Draper Software Process Model which uses a very similar process for explicit error creation and detection.[16] A causal diagram for the error and error detection process is shown in Figure 2. As shown, errors are created during a production stage, and removed by error discovery in subsequent phases.

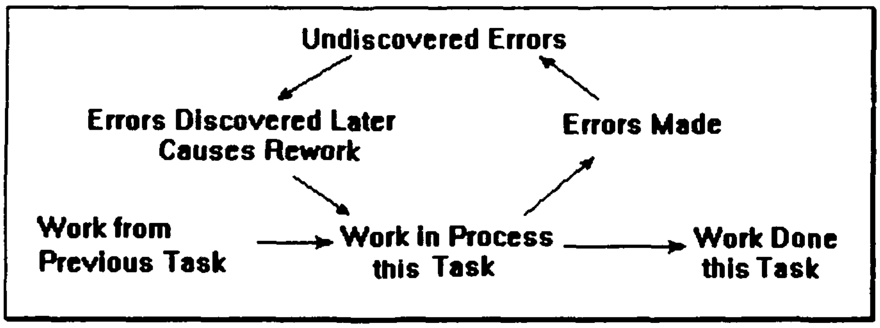


Figure 2
General Error Model

Since this system is duplicated for each task in the system, errors are explicitly tracked for each phase in the software process. Error creation occurs linearly, by multiplying the resource error rate by the amount of work done. Using a resource error multiplier allows us to better distinguish between skilled and unskilled resources. Error rates were derived using a pre-testing average of 50 Errors/KDSL (KDSL= thousands of lines of delivered source code) from Boehm,[17] Grady and Caswell,[18] and Abdel-Hamid's[19] figures as input. The following error creation rates were used in model calibration and testing (Table 2)

Table 2
Error Rates

Phase	Percent of Errs	Errors/KDSL
Requirements	18%	9.25
Design	52%	26.0
Coding	30%	15.0

Error removal and rework depends on the percent of errors detected in later phases. Since no phase detects 100% of errors, residual errors are a major output of the model. Further this system, through rework, accurately simulates the growth in project effort and size as errors are discovered and reworked later in the project. In essence, errors feed back into the system for correction as they are reworked.

An error is defined as "any flaw in the specification, design, or implementation of a product."[20] Default error rates may be set in the main module (EXAMPLE.CPP) for any resource. These default error rates are typically modified by the resource to factor in resource state information such as level of workforce burnout. Similarly, error detection may be adjusted both by adjusting the percent of errors detected in each phase and the error rework multipliers for each phase in the same way.

Effort Estimation

An important feature of the model is the effort estimation system. This system tries to determine how many resource-days of effort are needed to complete the project based on

current productivity rates and work waiting in all tasks. This estimation gives the user an estimate of how well the project is doing relative to a fixed schedule or budget, and also is used to calculate schedule pressure effects on the workforce (described under Resource Attributes below). This estimation is conceptually similar to Abdel-Hamid's work estimation system,[21] but in this model the estimation is explicit for all tasks in the model.

The estimation used in this model is a straightforward division of work that is waiting by current productivity rates in all tasks. Errors accumulated in each task are not included in the estimation. This reflects the fact that many managers do not include error estimates in their work estimates, because they are largely unknown to the manager.

The fact that *Task* objects in the model may apply conversion rates to work as it is done, and the fact that one *Task* may send its output to multiple *Tasks* in parallel complicates the work estimation process. Fortunately, a *Task* object can simply call the work estimation member functions for its list of next tasks to get an estimate of how long work in this task will take to complete. Since each *Task's* estimation function then calls the estimation function for its next tasks, the work estimation will mirror the actual work list, and produce an accurate estimation. If we repeat this process for all *Tasks* with work waiting in them, we get an accurate picture of the overall effort required to complete the project.

Resource Attributes

Since manpower is the primary resource in software development, this section will focus on the *People* class attributes rather than the general *Resource* class. Though the *Resource* class provides a generic framework for modeling other types of resources, the emphasis here is on manpower.

Resource Allocation

The model allows many instances of the same *Resource* class to be attached to a single process. This lets one flexibly model multiple work groups and their allocation to the tasks at hand. Since each resource can have unique attributes with respect to each task, it is relatively easy to model groups with varying ability, specializations, and attributes.

In our particular implementation, allocation is recorded in an *Allocation* object. *Allocations* link the resources to the tasks they work on. Each time step, these allocations are assessed and adjusted by the model's *Allocate*() member function. Since this function can be overridden using inheritance, it is possible for new models to be defined with alternative allocation strategies.

Currently the *Allocate()* function does an iterative allocation based on average work waiting for each task. The allocation function does the following: First, the average productivity for each task is *estimated and used* to determine the number of resource days required for each task. Next, for each resource, the function allocates resources based on the number of resource days required for each task the resource can perform. Finally, actual *Allocation* objects are created based on the allocations made. Other allocation schemes are possible, like allocating on an ability basis or even first come first served. This particular algorithm was chosen because it was similar to that which is used in the Draper model. Since we wished to do a direct comparison with the Draper model, this allocation scheme seemed most appropriate. Exploring the effects of allocation algorithms on the project was left as an open subject for future research.

Workforce Turnover

Any project of moderate length general has turnover in its workforce. Reasons include quitting, reassignment, and retirement. According to Abdel-Hamid, turnover in many software projects can run as high as 34%.[22] Our model defines turnover in terms of average retention time. Each *People* object has a retention time attribute which may be set when an object is

instantiated and adjusted during a run. As the model executes, the number of people in the resource is divided by the average retention time to determine the losses for a given step.

Communication Overhead

Probably the most significant factor affecting workforce productivity is the degree of communication overhead. The more people involved in a project, the more overhead experienced. This is because adding one person to a project increases the number of possible communication paths by the number of people already on the project. This means that overhead initially goes up as the square of number of people. Clearly this effect is bounded at some point, or large organizations would get no work done. We estimate the peak overhead to be approximately 60% of total work done. A graph of the relationship is shown in Figure 3. The communication overhead is factored into the productivity of each People object based on the total number of people assigned to the project.

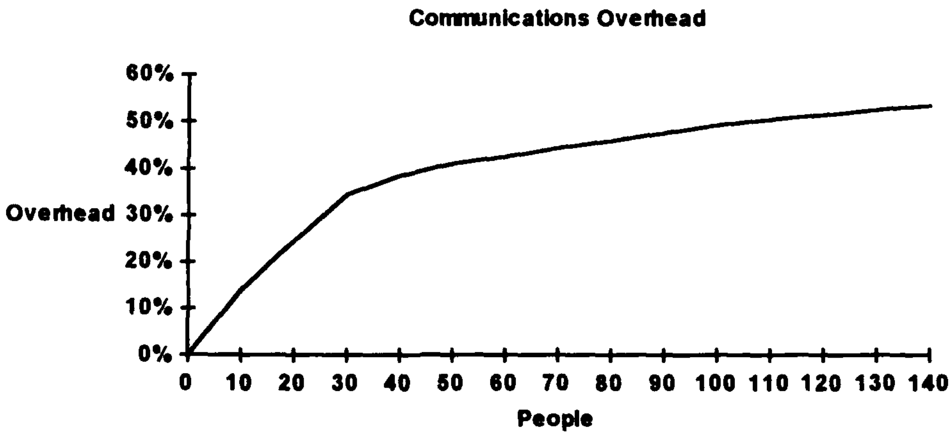


Figure 3
Communications Overhead

Training Effects

Often a project has a group of trainees. One of the principal advantages of the object-oriented approach to modeling is that groups of trainees can be modeled easily as separate

People objects. It is a well documented fact that trainees have lower productivity than experienced workers. Abdel-Hamid places the actual productivity of trainees to be between 0.33 and 0.5 that of an experienced worker.[23] To allow for variations, we implemented a simple system where the modeler can set both the initial productivity level and the average training time for an object. Each time step the productivity of the object is increased by the difference between 1.0 (full productivity) and the current training level divided by the average training time. This has the effect of slowly raising the productivity of the trainees over the average training time until they reach full productivity. Groups with different training periods may be modeled as different objects. This simple scheme is similar to Abdel-Hamid's approach to training, but our object-oriented approach allows for differences in the training of different work groups.

Overtime and Burnout

When people work an extended period of overtime, productivity and quality generally suffer. While such effects are minor for short periods, extended periods of overtime do have significant effects. This model uses a detailed overtime and worker burnout system based on Abdel-Hamid's work[24] as well as the Draper Software Process Model.[25] The assertion is that productivity falls and error generation increases as workers suffer the physical and emotional effects of extended overtime.

The implementation of these burnout effects is fairly straightforward. A variable keeps track of the percent of burnout for each *People* object. A second variable is used to indicate hours worked per week. As a workweek exceeds 40 hours per week, small amounts are added to the burnout value each time step in proportion to the level of overtime. As this burnout level accumulates, it is used as a factor in reducing productivity as shown in Figure 4 below. This lowering of productivity offsets the positive productivity effects of overtime. Further, as burnout increases, it is used as a multiplier in error generation. Though overtime initially adds to productivity, burnout will increase over time and eventually hurt both productivity and quality. In

the absence of overtime, the model reduces the burnout level so that workers freed from overtime will eventually return to their original productivity. Currently the time to burnout at a 60 hour workweek is set to 50 work days, and the burnout recovery period is set to 60 work days.

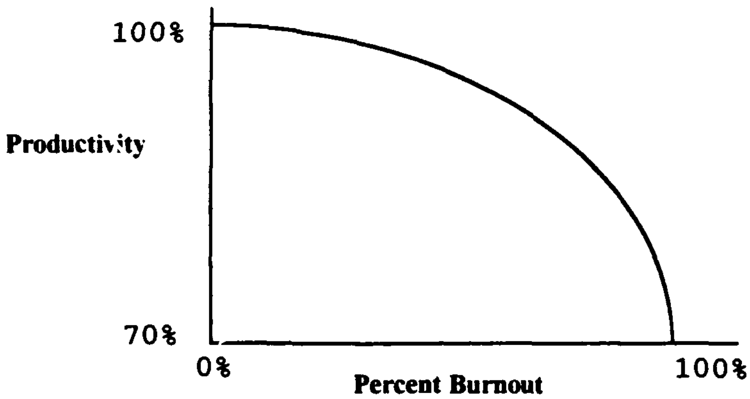


Figure 4
Productivity vs. Burnout

Schedule Pressure vs Productivity

Another well documented worker phenomenon is that schedule pressure can increase workforce productivity. Abdel-Hamid attributes this effect to the fact that workers generally have a certain amount of free or social time as part of their regular work day.[26] As pressure to perform increases, workers tend to spend less of their workday on social or personal activities and more time doing work. Sensitivity research done on the Draper model indicates that up to a 10-20% productivity gain can be observed.[27] In the model this productivity gain is modeled based on the estimated and actual mandays remaining in the system. From these manday estimates, the number of additional people to complete the project can also be estimated. This manpower shortfall is used as a basis for the modeling the productivity gain. The model will increase productivity due to schedule shortfall to make up for the shortfall until the limit of 20% additional productivity is reached.

Class Hierarchy for Model

Model Design

To develop a comprehensive object-oriented model for software development, a straightforward object-oriented design is used. At the highest level, modeling objects can be categorized as *Resources*, *Tasks*, and *Allocation* objects. These three classes represent physical resources, tasks, and allocation of resources in the development process. The common data elements in these three main objects led to formation of the *Value* class, the *Base* class, and the *NamedBase* classes shown in Figure 5. These classes are used primarily to maximize inheritance for the common elements in subsequent classes.

One advantage of object-oriented modeling becomes clear in this organization. Object-oriented languages have complete support for data abstraction. This feature is sorely lacking in current modeling tools. *Where a traditional dynamic model might* represent a task or resource as a aggregate group of variables, the object-oriented modeler can organize the model to represent real physical entities. Further, since the interfaces to an object are well defined in terms of its member functions, modeling objects can be developed independently. A traditional model is typically developed with global equations. An object-oriented model can be developed to take full advantage of data abstraction and data hiding.

Inheritance is also fully exploited using object-oriented modeling. The *Resource*, *Task* and *Allocation* classes all have common roots back to a simple data *Value* class. Further specific classes can be derived from more general classes. An example of this is shown in the derivation of the *People* class from the more generic Resource class. This leads to a high degree of reuse in the model.

A final advantage that was exploited in this model is the dynamic creation of objects. C++ allows one to create multiple instances of the same object. Thus, separate tasks in a project (requirements, design, coding, testing) are instances of the same *Task* class. Similarly multiple

resources are instances of the *People* class. This gives tremendous flexibility in adapting the model to new processes without modifying the underlying classes.

Overview of Model Classes

Class Definitions

The object-oriented software process model class hierarchy is diagrammed in Figure 5. Classes are described fully in Appendix A. The code for defining the classes is contained in Appendix B.

Model Class Hierarchy

The class hierarchy for the model was derived from the standard class library for the Borland object-oriented C++ compiler.[28] These standard classes are all derived from a single abstract base class called *Object*. The standard *class library* has many useful data structures pre-defined and implemented, letting us form *abstract* data structures without implementing them from scratch. For example, if we need an array of *Task* objects, we can define an *Object Array* and place our *Task* objects in it since the *Task* class was derived from the common Object class. The standard library has arrays, hash tables, dictionaries, strings, bags, ordered arrays, and many more objects that can be used to store our objects. Figure 5 illustrates the class hierarchy for the model. In Figure 5, standard library objects are shown in *Italics*.

The root class for our derived model is a class called *Value*. The *Value* class is used to represent a single variable in the model. The *Value* class has an initial value, current value, and a rate of change value. *Value* objects can be initialized, calculated, stepped, and restarted. The class is fully specified later in this document.

An *Allocation* class is derived from the *Value* class. This object represents allocation of a particular resource to a particular task. Allocations are recalculated each time step, as resources and tasks change. This class is detailed below.

The *Base* class is simply a *Value* object with an additional *Type* data member. The *Type* data member is used to represent the type of resource (people, machines, etc.) or task (Requirements, Design, Testing, etc.) represented by the object. The *Prod* class is the only class derived directly from the *Base* class. *Prod* objects are attached to each *Resource* to represent the nominal productivity of that resource on a particular type of task. *Prod* objects represent the nominal productivity as well as the nominal error rates and nominal error detection rates for a particular task. A *Rigor* level can also be associated with a *Prod* object to represent the rigor with which the resource approaches the assigned task.

The *NamedBase* class is used as a base for remaining classes. The *NamedBase* class adds a *Name* data member and unique *ID* identifier to the base class, letting us uniquely identify a particular task or resource. This allows two task objects of the same type (perhaps two testing objects) to be distinguished. The *Name* data member is used in reports on tasks and resources.

The *Task*, *Resource*, and *Model* classes are all derived from the *Named* base class. Each is a fairly complex object, fully detailed in the *Class* descriptions in Appendix B. From the *Resource* class, the *People* class is derived, which adds effects such as communication overhead and overtime burnout to the abstract resource definition.

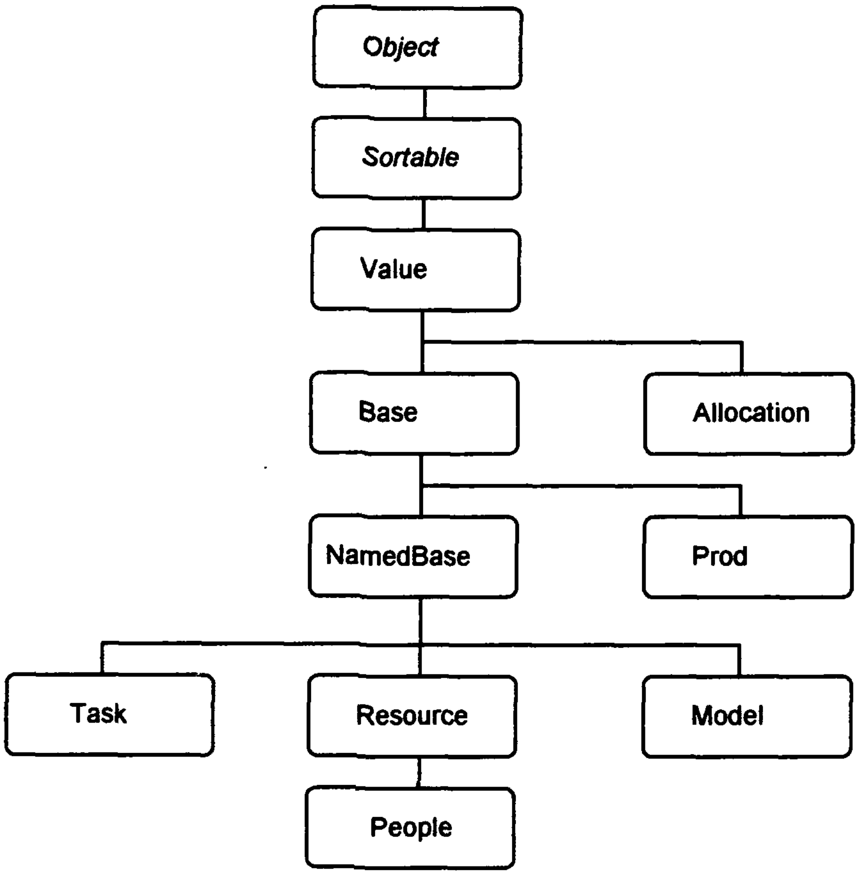


Figure 5
Model Class Hierarchy

Value Class

class Value: public Sortable; - The *Value* class is used as a root class for the model. It represents all time dependent values in the model. Each *Value* object can be reset to its initial value to restart the simulation. Each time step the *calculate()* member function is called to calculate the value for the next time step, then the *step()* member function is called to actually step to the new value.

Base Class

class Base: public Value; - The *Base* class is an abstract class used by later classes to identify a particular resource or task. The *Base* class inherits all of the functions and data

members from the *Value* class and adds a new data member called *type*. The *type* is used identify a particular group of resources (i.e. people) or a particular task (i.e. requirements work) in derived classes.

NamedBase Class

class NamedBase: public Base; - The *NamedBase* class inherits all of the data and member functions from the *Base* class and adds a unique identifier called *Id* as well as a *Name* for the object. It is an abstract class used to develop tasks and resources.

Productivity Class

class Prod: public Base; - The *Prod* class is used to define a resource's productivity for a particular task. Several productivity's are associated with each resource, defining that particular resource's base productivity for accomplishing a particular task. For example a group of programmers might have productivity's associated with the requirements, design, *coding and* documentation tasks. This class is derived from the *Base* class, so the value of the productivity as well as the task type are inherited directly from the base class.

Task Class

class Task: public NamedBase; - The *Task* class forms the basis for the production system. A task is any phase of the production process. *Tasks* can be chained together so that the output from one task becomes the input for another task. The main *Value* data member for the task keeps track of the work still to be done. Separate *Error* and *ConversionRate* data members are maintained to keep track of errors for this task and the conversion of work as it is processed.

Allocation Class

class Allocation: public Value; - The *Allocation* class is used to keep track of allocations of a particular resource to a task. *Allocations* are made each time step by the main model, and management of allocations is done by the main model class. The allocation objects simply store the resulting allocations. The class is derived from the *Value* class with the main *Value* specifying the number of resources allocated.

Resource Class

class Resource: public NamedBase; - The *Resource* class defines a generic resource and allows the user to attach productivities for different production tasks to the resource. Each resource has a name, type, and array of productivities for different tasks. *Resources* are derived from the *NamedBase* class with the main *Value* used to keep track of the number of resources available.

People Class

class People: public Resource; - The *People* class is derived from the generic resource class and inherits all resource characteristics. In addition the *People* class adds communication overhead and overtime effects to *People* resources.

Model Class

class Model: public NamedBase; - The *Model* class represents a model of a complete project including all tasks and resources assigned to it. Stepping the model forward in time effectively steps all of the objects associated with the model forward in time.

Class Hierarchy for Windows Displays

Overview

Class Definitions

The windows class hierarchy is shown in Figure 6, and described in the sections that follow the figure. Complete descriptions of the data members and member functions for the hierarchy is in Appendix A. Code for defining the hierarchy is in Appendix B.

Window Class Hierarchy

In additioı. to the *Model* classes, several Windows classes were derived from Borland's ObjectWindows™ library[29] to produce a primitive Microsoft Windows™ interface.[30] While the interface currently produces only simple tables and graphs showing values of key variables, it could easily be extended to allow the user to input variables much like the Microworlds™ interface lets us interact with the current software process model.[31]

The hierarchy is derived from the *TWindow* and *TFrameWindow* classes defined as part of Borland's ObjectWindows library. Borland library classes are shown in *italics* on the class hierarchy diagram, Figure 6. The *TWindowFrame* class provides a multi-document window suitable for displaying several subwindows and a menu. The *TWindow* class is then used to derive the sub-windows for the model, with each window displaying information about a particular portion of the model. *GraphWindows* also use two small classes called *GPoint* and *Graph* used to store the data necessary for a graph.

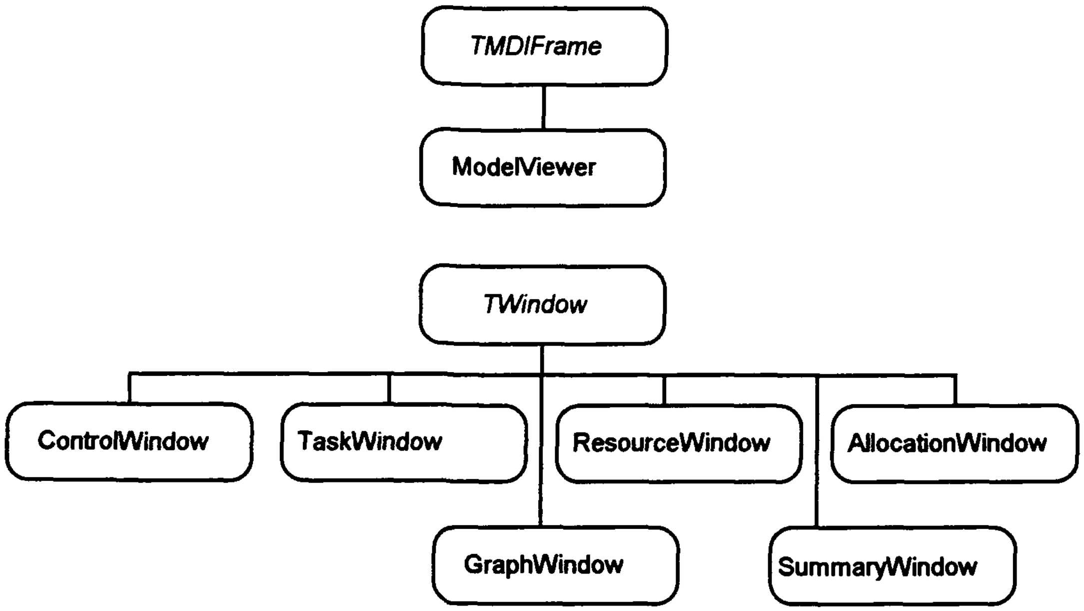


Figure 6
Windows Class Hierarchy

ModelViewer Class

class ModelViewer: public TMDIFrame; - The *ModelViewer* class forms the main multi-document windows class for the model. This class handles requests from the user (menu commands) and serves as a backdrop for the other windows.

ControlWindow Class

class ControlWindow: public TWindow; - The *ControlWindow* class displays a small window with four buttons on it. The buttons allow a user to step, restart, run continuously, and quit from the model. Pressing the corresponding button results in a message being sent to the main *ModelViewer* window where it is processed.

TaskWindow Class

class TaskWindow: public TWindow; - The *TaskWindow* class displays a list of all task names, the work to be done, and the errors generated for each task associated with a model.

ResourceWindow Class

class ResourceWindow: public TWindow; - The *Resource* window displays the name of each resource associated with the model along with the number of resources. The display is updated with each time step.

AllocationWindow Class

class AllocationWindow: public TWindow; - The *AllocationWindow* shows a list of all resource allocations currently active in the system. For each it shows the task the resource is allocated to and the number of resources allocated. The display is updated as the model is stepped.

SummaryWindow Class

class SummaryWindow: public TWindow; - The *SummaryWindow* shows a report of the key outputs of the model including mandays expended, size of project, and errors per thousand lines of code. The display is updated as the model is stepped.

GPoint Class

class GPoint: public Sortable; - The *GPoint* class is a simple class used to store points for a particular Value object in the model as the model is run. These points can then be displayed in a GraphWindow.

Graph Class

class Graph: public Value; - A *Graph* object represents the storage of a single Value object over time. *Graphs* are ultimately displayed as single data series lines on *GraphWindows*.

GraphWindow Class

class GraphWindow: public TWindow; - The *GraphWindow* class simultaneously displays multiple *Graph* objects in a window on the screen. This class takes care of drawing and scaling multiple data sets on the same graph.

Sample Model Structure

The sample model was created to demonstrate the capability of the new model as well as create a baseline for comparison v Draper's model and Boehm's COCOMO model. The sample model is structured as shown in Figure 7. Complete code for creation of the model is attached in the Appendix B as file EXAMPLE.CPP.

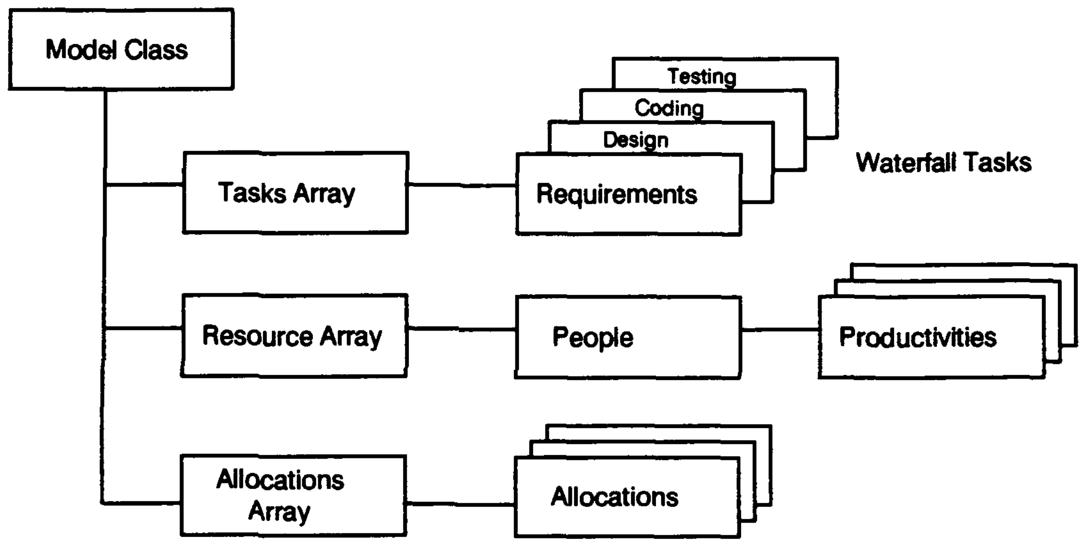


Figure 7
Sample Waterfall Model Structure

The waterfall model shown above closely duplicates the structure of the existing Draper STELLA™ model. However this new model could easily be extended to include more advanced structures. For example, by adding only a few lines to the EXAMPLE.CPP file one could create a second *People* object and populate it with specialized productivities. This might be used to model the role that a group of requirement and design specialists take in the process. Assuming the same model definitions and variables defined in EXAMPLE.CPP, here's the code for adding a second set of people that specialize in programming but do not do design well. Results from running this example are included in the section on *Calibration and Testing* later in this thesis.

```
#define PROGRAMMERS 101    // Unique resource type for new specialists

// Create new requirements people with seven specialists in the group

r = new People(m, "Programmers", PROGRAMMERS, 7.0, WILL_USE_OT);

// Make these new people specialize in programming by adding
//  to the programming ability and reducing the design productivity.
//  Make them do programming twice as fast and design half as fast

 r->add(* (new Prod(REQUIREMENTS,RQTS_RATE,RQTS_ERR_RATE,
             RQTS_ERR_MULT)) );
r->add(* (new Prod(DESIGN, DESIGN_RATE*.5, DSGN_ERR_RATE,
             DSGN_ERR_MULT)) );
r->add(* (new Prod(CODING, CODE_RATE*2.0, CODE_ERR_RATE,
             CODE_ERR_MULT)) );
```

Similarly if we wished to extend the waterfall to include initial requirements research or system integration, we need only know the relative productivity and error rates for these processes. We would create new *Task* objects for the new processes, add productivities to the appropriate resources, and chain the new tasks to the current tasks. More complex productivity models could be developed where the process forks into two production lines and then rejoins at some future production phase, since the generic *Task* object can easily send its output to more than one other task.

Sample Output

Below are several screen shots of different windows from the prototype object-oriented model running under Microsoft Windows™ version 3.1. Additional examples are in Appendix C.

Figure 8 shows the model in its initial state. There are 13.6 people (*Resource* Window) assigned to the sample model, with 32K waiting in the requirements task queue. The control panel with the step, quit and restart buttons is seen in the lower right corner. Total mandays expended is currently displayed in the summary window just to the left. All graphs are in their initial state.

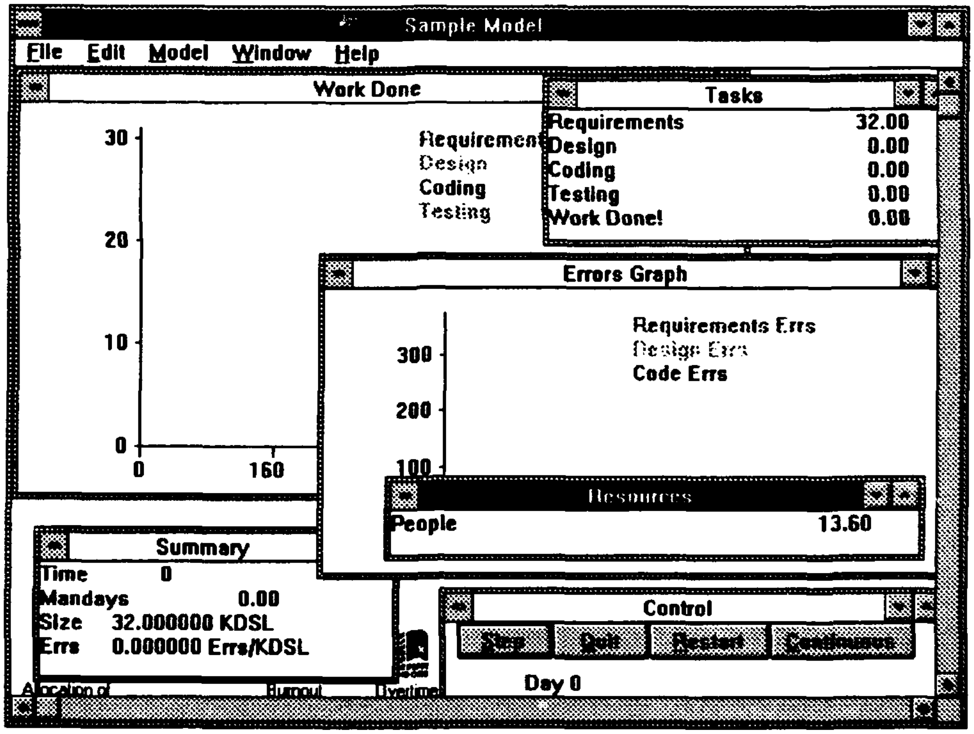


Figure 8
Initial Model Display

Figure 9 shows the model after it has stepped 200 days into the project. The *Tasks* window displays the work waiting in each task. The *Work Done* and *Errors* graphs show graphically the work being done and errors accumulating and being discovered in each task. The *Summary* window provides a summary of key project outputs.

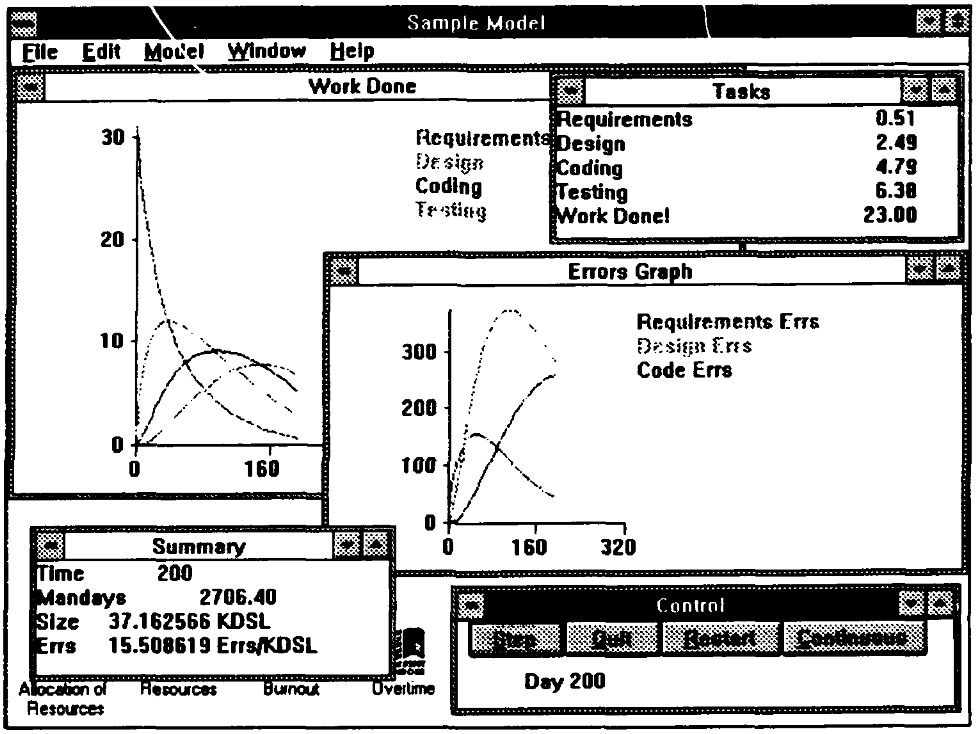


Figure 9
Running Model

Figure 10 shows the model at the completion of this 14 month project. The tasks are all completed, as all work currently resides in the *Work Done!* task. Residual errors for each task are displayed in the *Errors* graph. The summary window shows that 3549 mandays were expended on this original 32K project.

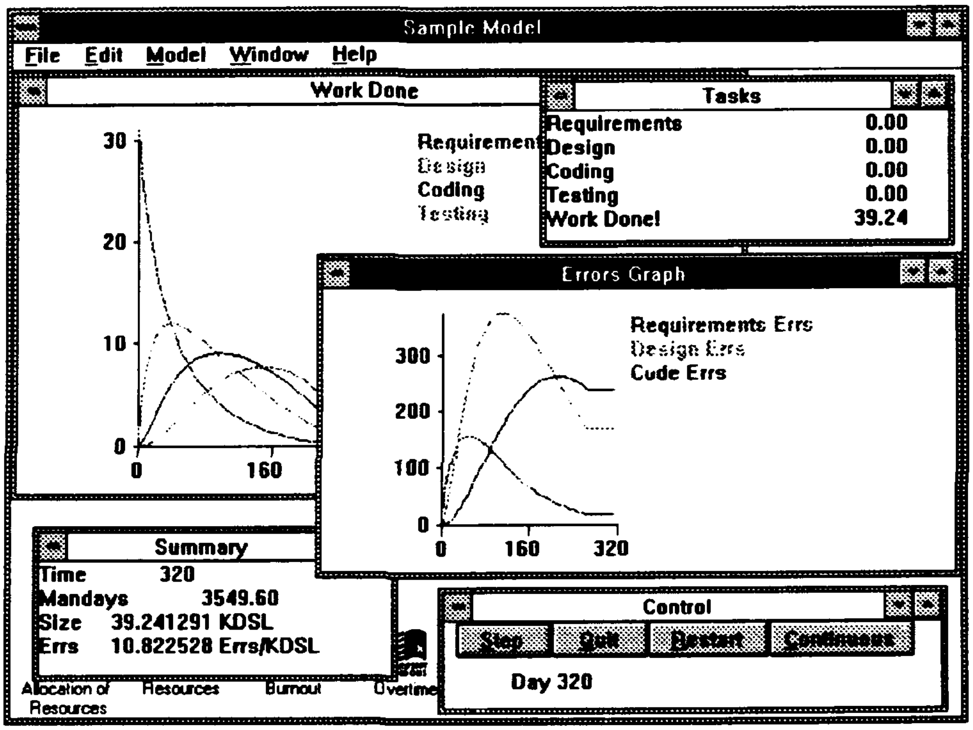


Figure 10
Completed Project

Model Calibration and Testing

Comparison with COCOMO and Draper Model

The completed model was set up to mirror the waterfall process as shown in Figure 1 earlier in this paper. *Task* objects were created to model each of the phases of the waterfall (i.e. requirements, design, coding, testing). Similarly a single *People* object was created, with nominal parameters, to model a homogeneous workforce. The automatic overtime feature was turned on for this group, but workforce attrition was turned off so that the workforce level would remain constant for the life of the project. Some initial model calibration was done to bring productivity in line with intermediate embedded COCOMO[32] for a 32K line project, resulting in a final productivity rate input of 13 Lines/day overall. COCOMO was chosen as the baseline, because it is one of the most widely used estimation models in the software industry.

Next the model was run over a wide range of projects, using the intermediate embedded COCOMO as a guideline. The COCOMO model provided the initial schedule and manpower estimated based on project size. Using these schedule and manpower settings, the model was run for project sizes ranging from 16 K to 1024 K lines, and resulted in the following comparative results. (Table 3)

Table 3
Object Model vs COCOMO

	COCOMO			*Object-Oriented Model*	
Size (KDSL)	**Schedule**	**Manpower**	**Mandays**	**Model Mandays**	**Pct Diff**
16.0	201.6	7.7	1,560.0	1,555.4	-0.29%
32.0	263.0	13.6	3,584.0	3,549.6	-0.96%
64.0	343.3	24.0	8,233.9	8,208.0	-0.31%
128.0	447.9	42.2	18,916.5	18,905.6	-0.06%
256.0	584.6	74.3	43,458.6	42,573.9	-2.04%
512.0	762.8	130.9	99,841.7	98,567.0	-1.28%
1,024.0	995.4	230.4	229,376.0	227,865.6	-0.66%

As evident in the results, the dynamic object oriented model performed very well over a wide range of project sizes, with an average deviation of -0.80% from COCOMO. This striking accuracy mirrors results from Draper's software process model, which consistently comes within 1% of COCOMO over the same project range.[33] This degree of accuracy is really quite striking when we compare the methods used. COCOMO is simply an exponential curve fit to a database of existing project data, with some linear work multipliers factored in. The object-oriented software process model, however, is a time-dependent simulation of a development process. What is significant in these results is that one can accurately simulate the non-linear relationships between resources, task, and time using dynamic modeling. Further, the accuracy of this object oriented model is very close to that of the Draper model on which it was based. This fact is further proof that an object oriented model has many of the advantages of that a non-object oriented model can.

Prototype Paradigm Experiment

As a demonstration of the flexibility of the model, a model based on the fast prototype paradigm was created and run. A fast prototype is a quick development stage that precedes the traditional requirements phase. A prototype is developed to help refine the requirements so that requirements and design of the final product can be done faster and with less errors.[34] We simplified the prototype phases by lumping them as one prototype task. This was done by creating a new prototype *Task* object and adding it to the model before the formal requirements phase. The model was populated with fictitious data using the assumptions stated below. We could find no reliable study of actual prototype projects to base this on, and decided to make this a demonstration of model flexibility rather than a test of modeling accuracy.

The following assumptions were used in adding a prototype stage to the traditional waterfall shown in Figure 1. These are all fictitious, and not designed to represent a real project.

1. All starting requirements were input to the prototype stage.
2. The prototype stage took approximately 15% of the overall project schedule.
3. The prototype stage allowed the requirements and design phases to proceed with half the error rate that they normally would have because errors were discovered during prototype.
4. The design phase was sped up from taking 20% of a traditional project to only 13% in the prototyped project. Similarly, the requirements phase was reduced from 15% of the project to 7%. Both speedups were due to work that could be taken from prototypes.
5. The prototype stage had error rates comparable to a normal requirements phase.

The intention overall was to make the prototype project and waterfall project be comparable in overall percent of scheduled time to complete. Though in reality a prototyped project might take longer to complete, we wanted to normalize the two to determine if the significant reduction in errors and time to complete the requirements and design phase would overcome the cost and errors of prototyping. This model is reflected in the final EXAMPLE.CPP file in Appendix B, with the conditional compile flag PROTO defined.

We used a baseline 32K project, with a 263 day schedule and 13.6 people assigned to it, just as in the original COCOMO calibrations from the previous section. The results are summarized in Table 4.

Table 4
Waterfall vs Prototype Paradigms

Paradigm	Mandays	Errors/KDSL
Waterfall	3584.0	10.81
Prototype	3413.6	9.37

As shown in Table 4, the prototype model, running with the assumptions made above, proved to be a cost-and-error-saving strategy. Apparently the savings from the prototype stage did overcome the additional errors and cost of implementing the stage. Please note that this is a fictitious example, so no real life conclusions should be drawn from it. Still, this example does an excellent job of underscoring the flexibility of the object-oriented approach. By changing a few lines in the EXAMPLE.CPP file we were able to add an entirely new phase to the model and run it. This kind of flexibility is not possible without major changes in non-object-oriented models such as the Draper software process model.

Multiple Resource Example

As a demonstration of a project with non-homogeneous resources, the model was run with a second *People* object of programming specialists, using code described in the previous section on *Sample Model Structure*. The scenario used was again a 32 thousand-line project, with a total of 13.6 people assigned, and overtime allowed. The workforce this time was divided into two objects. The first object had the same nominal 13 lines/day productivity used to calibrate the model against COCOMO, and consisted of 6.6 people. The second object had the same productivity as the first for all but the design and programming specialists and consisted of seven people. These "programming specialists" worked at twice the default speed during the coding phase, and only half the normal speed during design. Though this example is fictitious, the purpose was to demonstrate that non-homogeneous resources do make a difference in model output.

When the model was run with this additional *People* object, and the output compared to model output with a single resource object the following results were obtained (Table 5).

Table 5
Homogeneous vs Non-Homogeneous Resource Example

Paradigm	**Mandays**	**Errors/KDSL**
Homogeneous	3584.0	10.81
Non-Homogeneous	3413.6	10.83

Table 5 shows that the project with the homogeneous group of people took longer than the project with the non-homogeneous mix with programming specialists. The point is not that one is larger than the other, since the example was fictitious. The main point is that the two are different. Indeed the difference would probably be much larger if we used a more realistic mix of specialists in each task and included an allocation algorithm that would take resource skills into account. However, this limited test did demonstrate that multiple resources with different specialties can easily be added to the model, and that these resource specialties do affect the outcomes of a project.

Advantages and Disadvantages of the Object-Oriented Approach

Advantages

- The object-oriented approach allows for easy extension of the *Task* and *Resource* classes. The *People* class, for example, was derived from the generic *Resource* class by simply adding additional productivity and error factors associated with people.
- The object-oriented approach allows many types of processes to easily be modeled by simply chaining process objects together in different ways. For example, the waterfall life cycle model could be extended by adding a maintenance phase after the testing phase. Doing this using a non-object-oriented model would require almost a full redevelopment.
- The object-oriented approach can accurately model multiple resources with different production abilities. Further, specialized resources (i.e. analysts, programmers, designers, testers) can be modeled in a way that accurately reflects their specialization.
- The level of effort associated with tailoring the object-oriented model is significantly less than with a non-object-oriented approach. Currently someone tailoring the model will still require a degree of object-oriented knowledge, but that could be overcome if an interactive model-builder is built. Such a tool would take advantage of the fact that objects can be created and destroyed dynamically to let a user develop their own models based on pre-defined classes.

Disadvantages

- The introduction of resource specialization complicates the problem of effective resource allocation. While the sample *Model* class implements one solution by having each resource allocate to its tasks based on resource days required for the task, the model does leave open the option of developing alternative resource allocation strategies. This

is not a problem for a non-object-oriented approach since it assumed largely homogeneous resource pools whose productivity was based primarily on experience.

- Implementation of an object-oriented model requires more up-front design and planning. While traditional models allow global variable access, and unstructured modeling, object-oriented modeling requires a high degree of organization, with well defined interfaces between objects. As a result, some initial design work is needed to define an object-oriented model that is both functional and flexible.

Future Directions

The current object-oriented Software Process Model was designed as a proof of concept, with the focus primarily on the model, and not on the interface. As a result the user interface is quite primitive. In addition, the research has yielded some new, promising additions that could be implemented in future research.

Model Enhancements

Areas that could be improved upon include the following:

- **Alternative Resource Allocation Algorithms:** While the current implementation does accurately model the fact that managers generally make allocation decisions at a local level, many other allocation schemes are possible. A global allocation system based on worker ability would lead to better overall resource allocation in an organization.
- **Develop Generic Object-Oriented Modeling Tools:** One significant outcome of the research was a realization of the shortcomings of coding objects in a compiled language like C++. Though an object-oriented model is unusually powerful and flexible within the range of its usage, it would be nice if there was a language or tool specifically designed for object-oriented modeling. Clearly C++ and other object-oriented languages are outside the realm of many modeler's capabilities. In theory, an object-oriented modeling language or graphical tool could be developed to specifically support modeling applications. Such a tool might have pre-defined objects to represent various modeling constructs, as well as container objects that let a modeler handle multiple instances of an object. Separate methods for defining and instantiating runnable object would need to be defined.
- **Implement and Test Other Development Models:** The waterfall development cycle modeled here and in the original software process model is only one paradigm for

software development. We could easily generate other types of models (i.e. spiral, rapid prototype, etc.) using the building blocks from this object-oriented model. Though we have prototyped the structure of a rapid prototype paradigm, no industry data was available to serve as a fair comparison for the model. It would be most interesting to test these models against actual project data.

- **Develop Model using Object-Oriented Design Process:** Although the final product of this development effort is neatly organized, clearly improvements could be made in the structure of the model. It is likely that a modeler familiar with formal object-oriented design who approached the problem from scratch might take a different approach. For example, object attributes might better be stored in an array within the object so that generic *restart*(), *step*(), and *calculate*() functions can be written. Similarly, one might want to differentiate between the System Dynamic entities of stocks, flows, constants and converters, implementing each as a different object. In our model the differentiation is somewhat blurred as all are implemented using *Value* objects. The fact that we heavily used the Draper model for a basis may also have biased our design, since it was implemented in a traditional unstructured modeling language. Clearly developing an object-oriented model requires additional effort in planning and design.

Windows Interface Enhancements

The object of this thesis was to focus on the modeling aspects. As a result the windows interface for the model is very primitive, consisting primarily of simple tables that display critical model values while the model runs. Several enhancements could be made:

- **Improved User Interface:** To meet the time constraints imposed by a single semester thesis, a conscious decision was made to emphasize the model over the user interface. Though the present model does present informative graphs and tables in an interactive

Windows environment, the interface is somewhat awkward and unpolished. Another entire semester could be spent to design and implement a friendly interface.

- **Control of Decision Values:** It would be nice to be able to modify resource and task variables during a simulation run. This includes both initial project variables and time dependent decision variables. The model itself does not preclude changing variables in mid-project, but a full interactive interface must be developed. This would probably be implemented with popup dialogs for each resource and task class.
- **User Definable Reports, Tables and Graphs:** The user needs an easy way to be able to define their own reports, graphs and tables. This would probably be done from a menu or control panel selection. Currently the reports and graphs are instantiated in the model and can only be modified within the code.
- **Saved Variables:** The user may want to be able to save the state of the model at any point so they can come back to that project at a later date. This could easily be implemented by having each object write its current state to a file.
- **Interactive Model Development:** Since models can be created dynamically, it is feasible that an interactive model-builder could be developed. Such a tool would let the user choose from a set of resource and task types, assemble them in any order, seed them with appropriate values, and run the resulting model. This kind of tool would take a significant expenditure of resources to develop, but would bring the flexibility of the object-oriented model directly to users who have no knowledge of object-oriented programming.

Conclusion

This first implementation of an object-oriented software process model achieved the following:

- Established a base simulation engine for running a system dynamics model in the absence a commercial modeling language. This model may be stepped and restarted freely.
- Defined an abstract object-oriented class hierarchy for modeling typical software processes. This class hierarchy is capable of allocating multiple resource pools to arbitrarily chained tasks. Neither multiple task pools nor arbitrarily chained tasks could be implemented using current non-object-oriented modeling tools.
- Demonstrated the use of object oriented inheritance to derive specific objects from abstract classes. Defined a specific resource called *People* from the generic *Resource* class definition. The *People* class adds overtime and communication overhead multipliers to the generic *Resource* class definition. The generic *Task* class could also be extended by inheritance to include more complicated process elements.
- Designed, implemented and tested a primitive Microsoft Windows interface to the object-oriented software process model, including graphical output of key variables.
- Used the class hierarchy to model a classic waterfall software development. This sample development (EXAMPLE.CPP in Appendix B) was seeded with nominal productivity values and run to completion.
- Made a quantitative comparison between the object-oriented model and Boehm's COCOMO model, with surprising accuracy. The object-oriented approach achieved the same high degree of accuracy we have observed in calibration of the traditional Draper software process model against COCOMO over a wide range of project sizes.

- Made a comparison of the advantages and disadvantages of the object-oriented model when compared to the current implementation of Draper's Software Process Model using the non-object-oriented STELLA modeling language.
- Proposed areas for future enhancements to both the model and the Windows interface.

The object-oriented modeling technique overcomes many of the limitations of traditional modeling techniques. The high degree of accuracy and flexibility shown in this model makes this technique a promising new modeling approach for System Dynamics. The results of this project indicate that further research and development in object-oriented modeling shows great promise as a more structured technique for modeling complex dynamic systems. This base model can be extended to provide a more robust and flexible software process model.

Appendix A

Class Member Descriptions

Model Class Descriptions

Value Class

class Value: public Sortable; - The *Value* class is used as a root class for the model. It represents all time dependent values in the model. Each *Value* object can be reset to its initial value to restart the simulation. Each time step the *calculate()* member function is called to calculate the value for the next time step, then the *step()* member function is called to actually step to the new value.

Data Members

double value; - The current value of this variable for this time step.

double flowvalue; - The value of any flows to be added to this variable for the next time step.

double startvalue; - The initial value of this variable at the start of modeling. This variable is reset to this value when the restart() member function is called.

Member Functions

Constructor Value(double v); - Creates a new value with initial value equal to v.

Constructor Value(); - Creates a new value initialized to zero.

virtual double getValue() const; *- Returns the current value of this variable.*

virtual void initValue(double val); - Initializes this value and the startvalue to val.

operator double(); - Calls getValue() to return the current value.

void addFlow(double flow); - Adds the value flow to the current value when the next time step takes place. Used when a variable changes over time (like a stock).

virtual void paintTo(HDC hdc, PRECT r); - Windows function that paints the current value to a given device context at location specified by rectangle r.

virtual void restart(); - Sets the value back to its initial value (startvalue).

virtual void calculate(); - Dummy function used in inherited classes to perform calculations for the next time step.

virtual void step(); - Adds any pending flows to the actual value. Effectively steps the value forward one time step.

Base Class

class Base: public Value; - The *Base* class is an abstract class used by later classes to identify a particular resource or task. The *Base* class inherits all of the functions and data members from the *Value* class and adds a new data member called *type*. The *type* is used identify a particular group of resources (i.e. people) or a particular task (i.e. requirements work) in derived classes.

Data Members

IDTYPE type; - An integer used to identify the type of task or resource. Used in later classes that require ID's.

Member Functions

Constructor Base(IDTYPE t); - Creates a new base object with *type* set to t and an initial value of zero.

Constructor Base(IDTYPE t, double val); - Creates a new base object with *type* set to t and an initial value of val.

IDTYPE getType() const; - Returns the *type* data member of this object.

void setType(IDTYPE t); - Sets the *type* data member to be equal to t.

NamedBase Class

class NamedBase: public Base; - The *NamedBase* class inherits all of the data and member functions from the *Base* class and adds a unique identifier called *Id* as well as a *Name* for the object. It is an abstract class used to develop tasks and resources.

Data Members

static IDTYPE id_seed; - A shared seed used to give each NamedBase object a unique ID. Each resource and task will get a unique ID number based on this seed.

IDTYPE Id; - A unique ID for every task and resource in the system. Used to identify a particular task or resource.

LPSTR Name; - The name of the resource or task. A string.

Member Functions

Constructor NamedBase(LPSTR name, IDTYPE t, double val); - Creates a NamedBase object with *Name* set to name, of *type* t and with *value* val.

LPSTR getName() const; - Returns the *Name* of this object.

void setName(LPSTR s); - This function can be used to rename a NamedBase object.

IDTYPE getId() const; - Returns the unique Id associated with this object.

Productivity Class

class Prod: public Base; - The *Prod* class is used to define a resource's productivity for a particular task. Several productivity's are associated with each resource, defining that particular resource's base productivity for accomplishing a particular task. For example a group of programmers might have productivity's associated with the requirements, design, coding and documentation tasks. This class is derived from the *Base* class, so the value of the productivity as well as the task type are inherited directly from the base class.

Data Members

Value ErrorRate; - The error rate associated with this particular resource attempting this particular task. This is the nominal error rate, still subject to error multipliers associated with the resource.

Value ErrorDetMult; - The *error detection multiplier used in determining the difficulty of* error detection. This multiplier is usually set based on 70% of the project size.

Value Rigor; - The rigor with which this resource approaches this task. May be changed over time using the addRigor() member function.

Member Functions

Constructor Prod(IDTYPE t, double rate, double err_rate=0.0, double err_det_mult=1e6, double rigor=1.0); - Creates a new Prod object for task of type t, with production rate=rate, and error rate, detection, and rigor defaulted or set to provided values.

virtual double getValue() const; - Gets the effective productivity rate.

double getErrorRate() const; - Returns the current *ErrorRate* data member's value.

double getErrorDetMult() const; - Returns the *ErrorDetMult* data member's value.

double getRigor() const; - Returns the *Rigor* data member's value.

void addErrorRate(double v); - Increases the error rate at the next time step.

void addErrorDetMult(double v); - Increases the error detection multiplier at the next step.

void addRigor(double v); - Increases the rigor value at the next time step.

virtual void restart(); - Sets all values back to their initial value (startvalue).

virtual void calculate(); - Calculates the productivity values for the next time step..

virtual void step(); - Adds any pending flows to the actual values. Effectively steps the values forward one time step.

Task Class

class Task: public NamedBase; - The *Task* class forms the basis for the production system. A task is any phase of the production process. *Tasks* can be chained together so that the output from one task becomes the input for another task. The main *Value* data member for the task keeps track of the work still to be done. Separate *Error* and *ConversionRate* data members are maintained to keep track of errors for this task and the conversion of work as it is processed.

Data Members

Model *model; - The model this task is attached to. The task calls the addTask() member function of this model on startup to attach itself to this model.

Array *NextTasks; - This task will send its output to each task in the NextTasks array by calling their addWork() member functions. Tasks can be arbitrarily chained together in this way so that the output from one goes to several tasks or the output of several tasks goes to one particular task.

Value Errs; - This value maintains the total errors for this task.

Value ConvRate; This value stores the conversion rate for the task. This conversion rate is used to convert input units to output units. The default is 1.0.

double ResDays; - This is a temporary value used during resource allocation. The allocator calculates the number of resource days required to finish this task and stores the value here. The allocator then allocates resources to each task based on the resource days required by that task.

Value ErrReworkMult; - The error rework multiplier. Discovered errors are multiplied by this multiplier to determine the amount of work needed to correct an error.

Value PctErrDet; - The percent of errors from previous phases that may be detected in this phase. Used to determine error discovery for previous tasks.

Value WorkDone; - The amount of work accomplished on this task for the current phase. Used to determine work estimates in the next time step.

Member Functions

Constructor Task(Model *mod, LPSTR name, IDTYPE type, double work, double err_rework_mult=0.0, double pct_err_det = 0.0); - Creates a task with name and type specified and assigns work to this task. The task is attached to the task list for the model specified. The ErrReworkMult and PctErrsDet values are set based on the corresponding inputs to this function.

~Task(); - Destructor that deallocates the task when the model is done.

Array *getNextTasks(); - Returns the NextTasks data member.

double getConvRate(); - Returns the value of the ConvRate data member.

virtual Value *getErrors(); - Returns a pointer to the Errs data member.

double getResDays(); - Gets the number of resource days required for this task.

void add(Task &t); - Adds a new task to the NextTasks member, so that output from this task will go to the task specified.

void addWork(double w); - Adds new work to the work waiting on the next time step.

void setResDays(double v); - Sets the number of resource days required to the value specified.

double getErrorsDetected(); - Returns the portion of errors from previous tasks detected during this task.

double getPctErrsDet(); - Returns the value of the PctErrsDet data member.

void initConvRate(double conv_rate); - Sets the work unit conversion rate to the value specified.

void detach(Task &t); - Removes the specified task from the NextTasks list.

void EstimateResDays(double work_in); - Estimates the number of resource days it would take to complete work_in amount of work including all later tasks given current work rates. This function is the heart of the work estimation system.

virtual void restart(); - Resets all of the task's values to their initial state.

virtual void calculateTask(Resource &r, double num, Prod &p); - Calculates the work done and errors created for this task using the number of resources specified with the productivity specified.

virtual void step(); - Steps all values for this task to the next time step's values.

Allocation Class

class Allocation: public Value; - The *Allocation* class is used to keep track of allocations of a particular resource to a task. *Allocations* are made each time step by the main model, and management of allocations is done by the main model class. The allocation objects simply store the resulting allocations. The class is derived from the *Value* class with the main *Value* specifying the number of resources allocated.

Data Members

Resource *res; - Specified the resource being allocated.

Task *task; - The task this resource is allocated to.

Prod *prod; - The productivity of this resource for this task. Used during work done calculations for the task.

Member Functions

Constructor Allocation(Task *t, Resource *r, double num, Prod *p); - Creates an allocation object with data members as specified in the arguments.

Task *getTask(); - Returns the Task data member for this allocation.
Resource *getResource(); - Returns the resource (res) data member for this allocation.
Prod *getProd(); - Returns the productivity of this resource for this task.
virtual void calculateTask(); - Calls the calculateTask() member function of the assigned task with arguments taken from this allocation. Calculates the work done by this allocation of resources.

Resource Class

class Resource: public NamedBase; - The *Resource* class defines a generic resource and allows the user to attach productivities for different production tasks to the resource. Each resource has a name, type, and array of productivities for different tasks. *Resources* are derived from the *NamedBase* class with the main *Value* used to keep track of the number of resources available.

Data Members

Model *model; - The model object that this resource is attached to.
Array *Prods; - Array of productivities for this resource assigned to a variety of tasks. Productivities may be assigned or de-assigned using the add() and detach() member functions.

Member Functions

Constructor Resource(Model *mod, LPSTR name, IDTYPE type, double num); - Creates a new resource with name, number and type specified and attaches it to the named model object. The resource is initially created with no productivity objects assigned. These must be adding using the add() member function.
~Resource(); - Deallocates the Prod objects associated with this resource.

Array *getProds(); - Returns a pointer to the Prods array of productivity objects.
Prod &getProd(IDTYPE task_type); - Looks up the particular productivity object for tasks of type task_type.
virtual double getProdMult(); - Gets the current productivity multiplier associated with this resource. Nominal productivities from the Prods array are multiplied by this productivity multiplier when work done is calculated. The default member function simply returns 1.0.
virtual double getErrMult(); - Returns the error multiplier associated with this resource. Again the nominal error rate is multiplied by this multiplier when calculating errors created. The default member function returns 1.0.
void add(Prod &p); - Adds a productivity object to this resource, thus specifying the nominal productivity for this resource assigned to a particular task type.
void detach(Prod &p); - Removes a productivity object from the productivity list for this resource.

virtual void restart(); - Restarts the resource - assigning all values to their initial value.
virtual void calculate(); - Calculates all values associated with this resource.
virtual void step(); - Steps all values for this resource one time step forward.

People Class

class People: public Resource; - The *People* class is derived from the generic resource class and inherits all resource characteristics. In addition the *People* class adds communication overhead and overtime effects to *People* resources.

Data Members

static Value TotalPeople; - A data member shared between all people resources. This variable tracks the total number of people active on this project. It is used to calculate mandays expended and communication overhead effects.

Value HoursPerWeek; - The number of hours per week assigned for this group of people. Anything over 45 hours per week is considered overtime, and is subject to burnout effects.

Value Burnout; - The burnout level for this group of people. Ranges from 0.0 to 1.0.

Value TrainLevel; - The training level as a portion of the potential productivity of the group of people. This level will increase over the TrainDelay time until the workers reach full productivity. Range is 0.0 to 1.0.

Value TrainDelay; - The training delay. The time, on the average, that it takes for this group of people to reach full productivity if they start with a training level below 1.0.

Value RetentionTime; - The average retention time for this group of people. Workers will suffer attrition over this average time.

double WillUseAutoOT; - A calibration feature that allows the modeler to enable automatic management of overtime. Using the automatic feature, the model will automatically assign overtime to this work group to compensate for resource shortfalls in the model. A value of 1.0 turns this on, while 0.0 turns it off.

Member Functions

Constructor People(Model *m, LPSTR name, IDTYPE type, double num, double AutoOT=0.0, double train_level=1.0, double train_delay=1.0e8, double retention_time=1.0e8); - Creates a people object with name, type, and number of people as specified and attaches this object to the specified model object. Corresponding data members are set based on the optional inputs AutoOT, train_level, train_delay, and retention_time.

virtual double getCommLoss(); - Computes the current communications overhead. Used by the getProdMult() member function to compute overall productivity.

virtual double getOTMult(); - Computes the current overtime multiplier for this particular resource based on the burnout level and current HoursPerWeek. This function is also called by the getProdMult() member function.

virtual double getProdMult(); - Computes the productivity multiplier for this group of people taking into account the effects of burnout, overtime, training level, and communication losses.

virtual double getErrMult(); - Returns the error multiplier based on current overtime, and training level.

virtual double getTrainLevel(); - Returns the value of the current TrainLevel variable.

virtual double getRetentionTime(); - Returns the value of the RetentionTime variable.

virtual double getTrainDelay(); - Returns the value of the TrainDelay variable.

Value &getHoursPerWeek(); - Returns a reference to the HoursPerWeek variable.

Value &getBurnout(); - Returns a reference to the Burnout variable.

virtual void addFlow(double num_people); - This function can be used to add to the pool of existing workers.

virtual void restart(); - Restarts the resource - assigning all values to their initial value.
virtual void calculate(); - Calculates all values associated with this resource.
virtual void step(); - Steps all values for this resource one time step forward.

Model Class

class Model: public NamedBase; - The *Model* class represents a model of a complete project including all tasks and resources assigned to it. Stepping the model forward in time effectively steps all of the objects associated with the model forward in time.

Data Members

Array *tasks; - A collection of all of the tasks to be processed in this model.
Array *resources; - A collection of all resources assigned to this model.
Array *allocations; - A list of allocations of resources to tasks. This list is updated every time step.
Array *graphs; - A list of all of the graphs for the model which must be stepped each time step.
double CurTime; - The current time (in days) simulated by the model.
int stop; - The stop time for the simulation in days.
Value Mandays; - The number of mandays used for this project.
Value EstResDays; - The current estimate of resource days to compete the project based on work remaining and current work rates.
Value ResDaysRemain; - The actual number of resource days left until the schedule expires.
Value Schedule; - The current scheduled completion for this project in days.
Value TotalPeople; - The total number of people assigned to the project.

Member Functions

Constructor Model(LPSTR name, int stop_time); - Creates an empty model with no resources, tasks, or allocations attached. Sets the stop variable to the stop_time specified. Model will simulate a total of stop_time days.
~Model(); - Destructor for model - deallocates all resources, tasks and allocations.
void InitArrays(); - Internal function used to set up the task, resource and allocation arrays.
Array *getTasks(); - Returns the array of tasks for this model.
Array *getResources(); - Returns the array of resource for this model.
Array *getAllocations(); - Returns the resource allocations for this model.
Task &getTask(IDTYPE id); - Gets the task from the task array with identifier id.
Value &getMandays(); - Gets the mandays value associated with the model.
void addTask(Task &t); - Adds a new task to the task list of the model.
void addResource(Resource &r); - Adds a new resource to the resource list.
Value &getMandays(); - Returns a reference to the Mandays variable.
Value &getEstResDays(); - Returns a reference to the EstResDays variable.
Value &getResDaysRemain(); - Returns a reference to the ResDaysRemain variable.
int getCurTime(); - Returns the current time in days.
Value &getSchedule(); - Returns a reference to the Schedule variable.
void initSchedule(double sched_value); - Initializes the Schedule variable.

void setResDaysRemain(double v); - Sets the ResDaysRemain variable.

double getResourceShortfall(); - Returns estimated resource shortfall based on the current EstResDays and ResDaysRemain variables.

virtual double getEffWorkMult(); - Gets the effective productivity multiplier due to schedule pressure on the workforce. Called by People resource objects to determine additional productivity due to schedule pressure.

virtual void Allocate(); - Computes all allocations for this time step based on the work waiting for each task and the resources available. Results are put in the model's allocations array.

virtual void EstimateResDays(); - The function that Allocate() uses to estimate the number of resource days required to finish a project.

void flushAllocations(); - Flushes the current allocations from the allocations array so that the next time step's allocation can be computed.

virtual void restart(); - Restarts the model - assigning all values to their initial value.

virtual void calculate(); - Calculates all values associated with this model.

virtual void step(); - Steps all values for this model one time step forward.

virtual void stepModel(int n); - Calculates and steps the model forward n time steps.

Windows Modeling Classes

ModelViewer Class

class ModelViewer: public TMDIFrame; - The *ModelViewer* class forms the main multi-document windows class for the model. This class handles requests from the user (menu commands) and serves as a backdrop for the other windows.

Data Members

Model *model; - The main model object to be displayed. This is the model class defined previously in this paper.

int nsteps; - The number of steps to move forward each time the model is stepped. The default is 20 days.

LPSTR Name; - The name of the model - used to title the window.

ControlWindow *control; - Control window - lets user step, restart, and quit the model.

ResourceWindow *res; - Window to display all active resources.

TaskWindow *tasks; - Window to display all active tasks.

AllocationWindow *allocations; - Window to display resource allocations.

SummaryWindow *summary; - Window to display summary report.

Member Functions

Constructor ModelViewer(LPSTR name); Creates a new *ModelViewer window.*

virtual void SetupWindow(); - Function that creates all of the subwindows for the model (i.e. ControlWindow, ResourceWindow, etc...) and makes them displayable.

virtual void Refresh(); - Refreshes each subwindow display with the latest model values. This is called each time the model is stepped to update the display.

virtual void CMStep(RTMessage msg); - Menu command that steps the model forward nstep days.

virtual void CMRestart(RTMessage msg); - Menu command to restart the model - to day zero.

virtual void CMContinue(RTMessage msg); - Run the model continuously until the preset stop time is reached.

virtual void CMOK(RTMessage msg); - Tied to the quit button - this function quits out of the model when pressed.

int getSteps(); - Returns the number of steps (nsteps).

void setSteps(int n); - Sets nsteps to the number specified. The default is 20 days.

Model *getModel(); - Returns the Model data member.

ControlWindow Class

class ControlWindow: public TWindow; - The *ControlWindow* class displays a small window with four buttons on it. The buttons allow a user to step, restart, run continuously, and quit from the model. Pressing the corresponding button results in a message being sent to the main *ModelViewer* window where it is processed.

Data Members

ModelViewer *model; - Pointer to the main ModelViewer window.

Member Functions

Constructor ControlWindow(ModelViewer *m); - Creates a new ControlWindow object with parent model window m.

virtual void SetupWindow(); - Creates buttons in the window and activates them.

virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps); - Called by Windows to repaint the window as needed.

virtual void CMStep(RTMessage msg); - Steps the model forward one step when the Step button is pressed.

virtual void CMRestart(RTMessage msg); - Restarts the model when the restart button is pressed.

virtual void CMOK(RTMessage msg); - Quits from the model when the Quit button is pressed.

virtual void CMContinue(RTMessage msg); - Run the model continuously until the preset stop time is reached.

virtual void CMNSteps(RTMessage msg); - Will allow the user to reset the number of days per time step for the model.

TaskWindow Class

class TaskWindow: public TWindow; - The *TaskWindow* class displays a list of all task names, the work to be done, and the errors generated for each task associated with a model.

Data Members

Array *tasks; - A pointer to the tasks array for the model.

Member Functions

Constructor TaskWindow(ModelViewer *m); - Creates a new TaskWindow object as a subwindow of the ModelViewer m.

virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps); - Repaints the task window updating the display to reflect the latest tasks list.

ResourceWindow Class

class ResourceWindow: public TWindow; - The *Resource* window displays the name of each resource associated with the model along with the number of resources. The display is updated with each time step.

Data Members

Array *Resources; - A pointer to the model's resource list.

Member Functions

Constructor ResourceWindow(ModelViewer *m); - Creates a new ResourceWindow object as a subwindow of the ModelViewer m.

virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps); - Repaints the resource window updating the display to reflect the latest resource list.

AllocationWindow Class

class AllocationWindow: public TWindow; - The *AllocationWindow* shows a list of all resource allocations currently active in the system. For each it shows the task the resource is allocated to and the number of resources allocated. The display is updated as the model is stepped.

Data Members

Array *Allocations; - A pointer to the allocation list of the model.

Member Functions

Constructor AllocationWindow(ModelViewer *m); - Creates a new AllocationWindow object as a subwindow of the ModelViewer m.

virtual void Paint(HDC hdc, PAINSTRUCT _FAR &ps); - Repaints the allocation window updating the display to reflect the latest allocations list.

SummaryWindow Class

class SummaryWindow: public TWindow; - The *SummaryWindow* shows a report of the key outputs of the model including mandays expended, size of project, and errors per thousand lines of code. The display is updated as the model is stepped.

Data Members

ModelViewer *m; - A pointer to the *main modelviewer window.*

Member Functions

Constructor SummaryWindow(ModelViewer *m); - Creates a new SummaryWindow object as a subwindow of the ModelViewer m.

virtual void Paint(HDC hdc, PAINSTRUCT _FAR &ps); - Repaints the summary window updating the display to reflect the latest summary information for this particular run of the model.

GPoint Class

class GPoint: public Sortable; - The *GPoint* class is a simple class used to store points for a particular Value object in the model as the model is run. These points can then be displayed in a GraphWindow.

Data Members

double val; - The value of this point.

Member Functions

Constructor GPoint(double v); - creates a new point initialized to this value.

double getValue(); - Returns the value of this point.

Graph Class

class Graph: public Value; - A *Graph* object represents the storage of a single Value object over time. *Graphs are ultimately displayed as single data series lines on GraphWindows.*

Data Members

LPSTR name; - The name of the variable being graphed.

int count; - A count of the current time step used to determine when to store the next point.

int point_count; - An interval of how many steps to count in-between data measurement. Rather than store every data point value, this count may be set to store every 5 or 10 points of data for long projects.

double minvalue; - The minimum of all points - used in graph scaling.

double maxvalue; - The maximum value for all points- used to scale the graph.

Array *points; - The array of GPoint objects used to store individual data values.

Value *value; - A pointer to the actual Value object being graphed. Every point_count time steps this value is polled and stored as a new point in the graph.

Member Functions

Constructor Graph(Value *pval, LPSTR name, int pt_count=5); - Creates a new Graph object which will track the Value object pointed to by pval every point_count time steps.

~Graph(); - Destructor that deallocates the points associated with this graph.

LPSTR getName(); - Returns the name of this graph.

void AddPoint(double v); - Creates and adds a new point with value v.

virtual void step(); - Steps the graph by sampling the value for this graph if it is the appropriate time.

virtual void restart(); - Restarts this graph by freeing all current points.

GraphWindow Class

class GraphWindow: public TWindow; - The *GraphWindow* class simultaneously displays multiple *Graph* objects in a window on the screen. This class takes care of drawing and scaling multiple data sets on the same graph.

Data Members

Array *graphs; - Array of Graph objects to display in this window.

double minval, maxval; - Overall minimum and maximum values for all of the graphs being displayed. Used to scale axis for the graph window.

ModelViewer *modelv; - Pointer to the model viewer window.

Member Functions

Constructor GraphWindow(ModelViewer *m, LPSTR name); - Creates a new GraphWindow object with title equal to the name variable. The GraphWindow is created empty, until graphs are added using the AddGraph function.

~GraphWindow(); - Destructor for the GraphWindow which deallocates all Graphs.

void AddGraph(Value *val, LPSTR name, int point_count=5); - Creates a new Graph object and adds it to this GraphWindow.

virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps); - Paints all of the Graph objects to the window including axis and legend.

virtual void Refresh(); - Forces the current window to be repainted.
void LabelYAxis(); - Labels the Y axis based on the current X and Y values.

virtual void step(); - Steps all of the Graph objects forward one time step.
virtual void restart(); - Restarts all of the Graph objects to their initial state.

Appendix B

Source Code

MODEL.H

```
// Object Oriented Software Process Model
// Thesis Project by Bradley Smith

#define valueClass (classType) 50

// The Value class forms the basis for all other classes in the model.
//  A value represents any value in the model.
class Value : public Sortable {

  public:
   double value,  startvalue, flowvalue;
   Value(double v) { initValue(v); };    // Constructor
   Value() { initValue(0.0); };

   // Override object class functions - Inherit from Object
   virtual hashValueType hashValue() const;
   virtual classType isA() const { return valueClass; };
   virtual int isEqual(const Object& testobj) const
      { return(value == ((Value &)testobj).getValue()) ; };
   virtual int isSortable() const {return TRUE; };
   virtual char *nameOf() const {return "Value" ; };
   virtual void printOn( ostream& out) const
      { out << value; };
   virtual int isLessThan(const Object& testObj) const;


   // Main Value class methods
   virtual double getValue() const { return value; };
   virtual void initValue(double v) { value = v;
      startvalue = v; flowvalue = 0.0; };

    // Any value may be calculated, stepped or restarted
   virtual void step() { value = value+flowvalue; flowvalue=0.0;};
   virtual void calculate() { };
   virtual void restart() { initValue(startvalue); };
   operator double() { return(getValue()); };
    void addFlow(double v) { flowvalue += v; };
#ifdef WINDOWS
   // A Windows function that paints the value on a window
   virtual void paintTo(HDC hdc, PRECT r);
#endif
 };

// Derive some useful classes
typedef WORD IDTYPE;  // Identifier type used for model objects

// The Base class adds a Type member to the inherited Value class.
// This allows the type of resource or task to be identified
class Base : public Value
```

```
 {
 public:
   IDTYPE Type; // type of resource or task (i.e. Requirements)

   Base(IDTYPE t)
      { setType(t); };
   Base(IDTYPE t, double v)
      { setType(t); initValue(v);};

    // Functions for manipulating the new type data member
   void setType(IDTYPE t) { Type = t; };
   IDTYPE getType() const { return Type; };
  };

// The NamedBase class adds a unique ID member as well as a Name to the
//  base class.  Resources and Tasks are derived directly from this
class
class NamedBase: public Base
 {
 public:
   // A shared seed which generates unique identifiers for objects
   static IDTYPE Id_seed;

    // The new Name and Id data members
    IDTYPE Id;
   LPSTR Name;

   NamedBase(LPSTR n, IDTYPE t, double v) : Base(t)
      { Name = n; Id = ++Id_seed; initValue(v); };

   // New functions for manipulating the name and id.
   LPSTR getName() const { return Name; };
   void setName(LPSTR s) {Name= s; };
   IDTYPE getId() const { return Id; };

 };

_CLASSDEF(Resource)

 // The productivity (Prod) objects are attached to resources to
 //  specify productivity and error rates for a particular resource
 // working on a particular task.
class Prod: public Base
 {
   // Main variables are the error rate, error detection time, and rigor
   // The Value member (inherited) stores the actual productivity
   //  and the Type member (inherited) keeps track of the task type.
   Value ErrorRate, ErrorDetMult, Rigor;

 public:
   // Constructor
```

```
   Prod(IDTYPE t, double rate, double err_rate=0.0,
   double err_det_mult=1e6, double rigor=1.0) : Base(t)
   { ErrorRate.initValue(err_rate);
ErrorDetMult.initValue(err_det_mult);
      initValue(rate); Rigor.initValue(rigor); };

  // Data member access functions
  double getErrorRate() const { return ErrorRate.getValue(); };
  double getErrorDetMult() const {return ErrorDetMult.getValue(); };
  double getRigor() const { return Rigor.getValue(); };
  virtual double getValue() const { return Value::getValue()/getRigor();
};
  void addErrorRate(double v) { ErrorRate.addFlow(v); };
  void addErrorDetMult(double v) { ErrorDetMult.addFlow(v); };
  void addRigor(double v) { Rigor.addFlow(v); };

  // Functions for controlling the Prod calculations and time step.
  virtual void step()
   { Base::step(); ErrorRate.step(); ErrorDetMult.step();
      Rigor.step(); };
  virtual void calculate()
   { Base::calculate(); ErrorRate.calculate();
      Rigor.calculate(); ErrorDetMult.calculate(); };
  virtual void restart()
   {Base::restart(); ErrorRate.restart(); ErrorDetMult.restart();
      Rigor.restart();};
  };

_CLASSDEF(Model)

// The resource class models a generic resource.  Each resource has
//  one or more productivity (Prod) objects attached that specify the
//  productivity of the resource for a specific task.
class Resource: public NamedBase
 {

public:
 Model *model;
 Array *Prods;    // Array of productivity

 // Constructor
 Resource(Model *m,LPSTR n,IDTYPE type, double num);
 ~Resource() { delete Prods; };  // Resource destructor

 // Data member access functions
 Prod &getProd(IDTYPE task_id);
 void add(Prod &t) { Prods->add(t); };
 void detach(Prod &t) { Prods->detach(t); };
 Array *getProds() { return Prods; };

 // Production and error multipliers - overridden later when we
```

```
 //  implement specific resources (like the people class)
 virtual double getProdMult() { return 1.0; };
 virtual double getErrMult() { return 1.0; };

 // Time dependent calculate(), step() and restart() functions.
 virtual void step();   // Need to step all prods !
 virtual void calculate();
 virtual void restart();      // Reset all resources

 };


// The Task class defines an element or phase of the software process.
// Each task has a type associated with it (i.e. requirements).  Tasks
// may be arbitrarily chained together using the add function.  Tasks
// also may have error pools (Errs) and a unit conversion process
// (ConvRate) associated with them.


class Task: public NamedBase
 {
   // The NextTasks array defines all tasks that occur immediately
    //   after this one.  Output from this task goes to all of these.
   Array *NextTasks;
    Model *model;
   Value ConvRate; // Unit conversion rate for this task
   Value Errs;     // Errors accumulated for this task.
   Value ErrReworkMult; // Error rework multiplier
    Value PctErrsDet;         // Percent errors detected

   Value WorkDone;  // Work done in this phase

   double ResDays;       // Temporary value used during resource
allocations
    double TotRes;       // Total resources assigned to task

  public:
   // Construcor
   Task(Model *mod,LPSTR name, IDTYPE type, double v,
      double err_mult = 0.0, double pct_det = 0.0);
   ~Task() { delete NextTasks; };     // Destructor

   // Data member access functions
   double getErrorsDetected();       // Return portion of errors detected
   Array *getNextTasks() { return NextTasks; };
   virtual Value *getErrors() { return &Errs; };
   double getConvRate() { return ConvRate.getValue(); };
   double getPctErrsDet() { return PctErrsDet.getValue(); };
   void initConvRate(double v) { ConvRate.initValue(v); };
   void setTotRes(double v) { TotRes = v; };
   void addTotRes(double v) { TotRes += v; };
```

```
   double getTotRes() { return TotRes; };

   // Add and remove tasks from the NextTasks list
   void add(Task &t) { NextTasks->add(t); };
   void detach(Task &t) { NextTasks->detach(t); };

   void addWork(double v) { addFlow(v); };

    // Time dependent calculation and stepping functions
   virtual void step() { NamedBase::step(); ConvRate.step();
      Errs.step(); ErrReworkMult.step(); PctErrsDet.step();
      WorkDone.step(); WorkDone.addFlow(-WorkDone.getValue());};
   virtual void calculateTask(Resource &r, double num, Prod &p);
   virtual void restart()
      { NamedBase::restart(); ConvRate.restart(); Errs.restart();
            ErrReworkMult.step(); WorkDone.restart();};

   // These funtcions used to calculate resource allocations by
    //  calculating the resource days required to complete this task.
   void setResDays(double v) { ResDays = v; };
   double getResDays() { return ResDays; };
   double EstimateResDays(double in); // Estimate resource days for work
remaining
};

// Allocation objects are used to specify allocation of a
//  particular resource to a particular task.  Resources
//  are reallocated by the main model class every time step.
class Allocation: public Value
 {
  Task *task; // Task to allocate to
  Prod *prod; // Productivity of resource allocated
  Resource *res; // Resource allocated
 public:
   Allocation(Task *t, Resource *r, double num, Prod *p)
      { task = t; prod = p; res = r; initValue(num); }; // Constructor

    // Member access functions
   Task *getTask() { return(task); };
   Prod *getProd() { return(prod); };
   Resource *getResource() { return(res); };

    // Fuction to actually apply this allocation to the task.
   virtual void calculateTask()
      { task->calculateTask(*res,getValue(), *prod); };
 };

#ifdef WINDOWS
_CLASSDEF(GraphWindow)
#endif
// The Model class serves as a master for the other objects.  A list
```

```
//  of tasks, resources, and allocations is maintained by this class.
//  Further the Model class recalculates allocations each time step.
class Model: public NamedBase
 {
 public:
  // Set of tasks, resources, and allocations for this model
  Array *tasks, *resources, *allocations, *graphs;
  int CurTime;  // The current time in days
  int stop; // Simulation time in days
  Value Mandays;   // Mandays expended to date on the project
  Value EstResDays;     // Estimate of mandays required to finish
  Value ResDaysRemain; // Actual mandays left in schedule
  Value Schedule; // Current schedule
  Value TotalPeople; // Total of all people last time slice

 // Constructor and destructor functions for the model
 Model(LPSTR name, int stoptime): NamedBase(name, 0, 0.0)
   { CurTime = 0; InitArrays(); stop = stoptime; restart();};
 ~Model() { delete tasks; delete resources; delete allocations;
   delete graphs; };


 // InitArrays initializes all of the arrays for the model.
 void InitArrays() { tasks = new Array(6,0,3); tasks-
>ownsElements(TRUE);
   resources = new Array(3,0,2); resources->ownsElements(TRUE);
   allocations = new Array(6,0,3); allocations->ownsElements(TRUE);
   graphs = new Array(6,0,3); graphs->ownsElements(TRUE); };

 // Member data access functions
 Array *getTasks() { return tasks; };
 Array *getResources() { return resources; };
 Array *getAllocations() { return allocations; };
 void flushAllocations() { allocations->flush(TShouldDelete::Delete); };
 Task &getTask(IDTYPE id);
 Value &getMandays() { return Mandays; };
 Value &getEstResDays() { return EstResDays; };
 int getCurTime() { return CurTime; };

 Value &getSchedule() { return Schedule; };
 void initSchedule(double v) { Schedule.initValue(v); };
 Value &getResDaysRemain() { return ResDaysRemain; };
 void setResDaysRemain(double v)
    { ResDaysRemain.addFlow(v-ResDaysRemain.getValue()); };
 double getResourceShortfall();     // Resources required to finish on
time
 virtual double getEffWorkMult() // Effective work multiplier due to
pressure
   { return MIN((1+getResourceShortfall()/TotalPeople.getValue()),
      1.2); };
```

```
 // Assign newly created tasks or resources to the project
 void addTask(Task &t) { tasks->add(t); };
 void addResource(Resource &r) { resources->add(r); };
#ifdef WINDOWS
 void addGraph(GraphWindow &g);
#endif

 // Control the project time steps
 virtual void step(); // Steps model one day forward
 virtual void stepModel(int n);  // Calculates and steps model n days.
 virtual void calculate();
 virtual void restart(); // Restart entire model

 // Reallocates resources to tasks based on work waiting
 virtual void Allocate();
 virtual void EstimateResDays(); // Estimate resource days required for
project.
   // Gets called by Allocate after allocation calculations
 };


// The People class is a special instance of Resource - People adds
//  overtime burnout and communication loss to resource calculations.
class People: public Resource
 {
 Value HoursPerWeek;    // Hours to work per week
 Value Burnout;               // Burnout percentage
 Value TrainLevel;     // Training level
 Value TrainDelay;            // Training delay
 Value RetentionTime;   // Worker retention time
 double WillUseAutoOT;  // Automatic Overtime Feature

public:
 People(Model *m,LPSTR n, IDTYPE type, double num,double AutoOT=0.0,
   double train_level=1.0, double train_delay=1.0e8,
      double retention_time = 1.0e8)
   :Resource(m,n,type,num)
   { model->TotalPeople.initValue(model->TotalPeople.getValue()+num);
   HoursPerWeek.initValue(40.0);
   TrainLevel.initValue(train_level);
   TrainDelay.initValue(train_delay);
   RetentionTime.initValue(retention_time);
   WillUseAutoOT = AutoOT;};   // Constructor

 // Functions for accessing data members
 virtual void addFlow(double v)
  { Resource::addFlow(v); model->TotalPeople.addFlow(v); };
 virtual void setHoursPerWeek(double hours)
   { HoursPerWeek.addFlow(hours-HoursPerWeek.getValue());};
 virtual double getCommLoss();      // Communication losses
 virtual double getTrainLevel() { return TrainLevel.getValue(); };
```

```
virtual double getRetentionTime() { return RetentionTime.getValue(); };
virtual double getTrainDelay() { return TrainDelay.getValue(); };
virtual double getOTMult(); // Overtime multiplier
Value &getHoursPerWeek() { return HoursPerWeek; };
Value &getBurnout() { return Burnout; };

// Productivity and error multipliers overridden from Resource class
virtual double getProdMult()       // Productivity multiplier
{ return((1-getCommLoss())*getOTMult()*getTrainLevel()
     * model->getEffWorkMult()); };
virtual double getErrMult() { return((1.0+0.5*Burnout.getValue())
  *(2.0-getTrainLevel())); };

// Functions to calculate, step and restart the model
virtual void calculate();
virtual void step() { Resource::step(); Burnout.step();
  HoursPerWeek.step();
  TrainLevel.step(); TrainDelay.step();
  RetentionTime.step();};
virtual void restart() { Resource::restart(); HoursPerWeek.restart();
  Burnout.restart();
  TrainLevel.restart(); TrainDelay.restart();
  RetentionTime.restart();};

};
```

MODWIN.H

```
// Model Windows Interface -- derived from objectwindows
// classes - Brad Smith, Thesis Project, Boston University

// Global variables for the size of the default text font
extern int h_txt, w_txt;

_CLASSDEF(ModelViewer)

extern ModelViewer *pModelViewer;

// The ControlWindow class displays buttons the user can press to
//  step, restart and quit the model.
class ControlWindow: public TWindow
 {
 ModelViewer *model; // A pointer to the ModelViewer to step
public:
 ControlWindow(ModelViewer *m);

 // Standard functions to setup and paint the window
 virtual void SetupWindow();
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);

 // Command functions activated by pressing buttons
 virtual void CMNSteps(RTMessage msg)=[ID_FIRST+R_NSTEPS];
 virtual void CMStep(RTMessage msg) = [ID_FIRST+CM_STEP];  // Step model
 virtual void CMRestart(RTMessage msg) = [ID_FIRST+CM_RESTART];  //
Restart
 virtual void CMOK(RTMessage msg) = [ID_FIRST+IDOK]; // Quit model
  // This Continue selection runs the model to completion
 virtual void CMContinue(RTMessage msg) = [ID_FIRST+CM_CONTINUE];
 };

// The AllocationWindow class displays the resource allocations for
//  the project including how many resources are dedicated to each task.
class AllocationWindow:public TWindow
 {
 Array *Allocations;
public:
 AllocationWindow(ModelViewer *m);
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);
 };

// The ResourceWindow displays all of the resources available
class ResourceWindow: public TWindow
 {
 Array *Resources;
public:
 ResourceWindow(ModelViewer *m);
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);
```

```
 };

 // The SummaryWindow displays summary of key variables
class SummaryWindow: public TWindow
 {
 ModelViewer *m;
public:
 SummaryWindow(ModelViewer *m);
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);
 };

// The TaskWindow class displays the work waiting and errors for each
//  task associated with the project.
class TaskWindow: public TWindow
 {
 Array *Tasks;
public:
 TaskWindow(ModelViewer *m);
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);
 };

// The ModelViewer class serves as the main frame window for the model.
//  All other model displays as well as the main menu are subwindows of
//  this model frame.
class ModelViewer: public TMDIFrame
 {
 public:
  int nsteps;  // Number of days per user time step
  Model *model;   // Actual Software Process Model (see model.h)

  // The following are displayed windows
  ControlWindow *control;
  ResourceWindow *res;
  TaskWindow *tasks;
  AllocationWindow *allocations;
  SummaryWindow *summary;

  // The Name member specifies the name to be displayed at the top
  //   of the frame window.
  LPSTR Name;

  ModelViewer(LPSTR name); // Constructor function

  virtual void SetupWindow(); // Initializes child windows
  virtual void Refresh(); // Forces all child windows do redisplay

  // CMStep is a menu selection (and button) that steps the model
  virtual void CMStep(RTMessage msg) = [CM_FIRST+CM_STEP];
  // CMRestart() is a menu selection and button that restarts the model
  virtual void CMRestart(RTMessage msg) = [CM_FIRST+CM_RESTART];
  // This Continue selection runs the model to completion
```

```
  virtual void CMContinue(RTMessage msg) = [CM_FIRST+CM_CONTINUE];

  // Using getSteps() and setSteps() the number of days per user step
  //   can be set.
  int getSteps() { return nsteps; };
  void setSteps(int n) { nsteps = n; };

  // Other classes can use the getModel() function to access the model.
  Model *getModel() { return model; };
 };

// Graph points - get put in point array for graphs
class GPoint: public Sortable
 {
public:
 double val;
 GPoint(double v) { val = v; };
 double getValue() { return val; };
 virtual hashValueType hashValue() const { return 0; };
 virtual classType isA() const { return valueClass; };
 virtual int isEqual(const Object& testobj) const
      { return(val == ((Value &)testobj).getValue()) ; };
 virtual int isSortable() const {return TRUE; };
 virtual char *nameOf() const {return "GPoint" ; };
 virtual void printOn( ostream& out) const
      { out << val; };
 virtual int isLessThan(const Object& testObj) const
   { return( ((GPoint &) testObj).val == val); };
 };

 // The Graph class is an autoscaling graph
 class Graph: public Value
  {
 public:
  LPSTR name;
  int count, point_count;     // Used for saving every N points
  double minvalue, maxvalue;
  Array *points;
  Value *value;
  Graph(Value *val, LPSTR nam, int np = 5)
   { minvalue=0.0; maxvalue=0.01; value=val; name = nam;
   points = new Array(5,0,1); points->ownsElements(TRUE);
   point_count = np; count = 0; };
  ~Graph() { delete points; };
  LPSTR getName() { return name; };
  void AddPoint(double v);
  virtual void step();
  virtual void restart() { points->flush(TShouldDelete::Delete); };
  };

 // The GraphWindow class - a window with multiple graphs on it
```

```
class GraphWindow:public TWindow
 {
public:
 Array *graphs;
 double minval, maxval;
 ModelViewer *modelv;

 GraphWindow(ModelViewer *m, LPSTR name);
 ~GraphWindow() { delete graphs; };

 // Step the graphs forcing each to capture the current point
 virtual void step();
 virtual void restart();
 virtual void Paint(HDC hdc, PAINTSTRUCT _FAR &ps);
 virtual void Refresh() { InvalidateRect(HWindow, NULL, TRUE);};
 void LabelYAxis(HDC hdc, RECT &g);
 // Add a new value to the graph
 void AddGraph(Value *val, LPSTR name, int np=5);
 };

// Select a color for multiple color options
// Sets both line and pen color...
void SelectColor(HDC hdc, int i);
```

MENU.H

```
// Menu Constants for model - from menu resource file
// by Bradley Smith

#define CM_STEP 500
#define CM_RESTART 501
#define CM_CONTINUE 502
```

RES.H

```
// Resources for model - from Windows resource file
// By Bradley Smith
#define R_NSTEPS 101
```

SPM.H

```
// Main header file for object oriented software
//  process model - Bradley Smith

// UNDEF WINDOWS to turn off windows functions in classes
#define WINDOWS
#define WIN30

#define TRUE 1
#define FALSE 0

#define MIN(x,y) ((x)<(y) ? (x): (y))
#define MAX(x,y) ((x)<(y) ? (y) : (x))

#include <stdio.h>
#include <owl.h>
#include <shddel.h>
#include <strng.h>
#include <array.h>
#include <button.h>
#include <math.h>
#include "res.h"
#include "menu.h"
#include "model.h"
#include "modwin.h"
```

EXAMPLE.CPP

```
// A modeling example - the waterfall software process model
// Independent Study Project - by Bradley Smith

#include <spm.h>

// Globals
ModelViewer *pModelViewer;    // Main Model Window

// First derive a Windows application from the default
// TApplication ObjectWindows class
class ModelApp : public TApplication
 {
public:
 ModelApp(HANDLE hInst, HANDLE hPrev, LPSTR cmdline,
   int CmdShow) : TApplication ("Software Process Model",
      hInst, hPrev, cmdline, CmdShow) {};

 virtual void InitMainWindow();
 };

// Create a new model derived from the standard Model class
//   that is initialized using InitModel();
class SampleModel: public ModelViewer
 {
public:
 SampleModel():
   ModelViewer("Sample Model") { InitModel();};
 virtual void InitModel();
 };

// Set the SampleModel up as the main window for our Windows
//   Application
void
ModelApp::InitMainWindow()
 {
 pModelViewer = new SampleModel();
 MainWindow= pModelViewer;
 };

// Constants used to identify task types
#define REQUIREMENTS 1
#define DESIGN 2
#define CODING 3
#define TESTING 4
#define DONE 5

// Initial problem parameters
#define SIM_TIME 320          // Simulation time in days
#define INIT_SCHEDULE 263   // Schedule in days
```

```
#define INITSIZE 32.0          // Size of the problem (KDSL)
#define INITPEOPLE 6.6         // Number of people
#define WILL_USE_OT 1.0        // Willingness to use overtime

// Production controls
#define NOM_RATE 13.0          // Nominal Lines per day
#define RQTS_PCT 15.0        // Percent time spent in each phase
#define DESIGN_PCT 20.0
#define CODE_PCT 40.0
#define TEST_PCT 25.0
#define RQTS_RATE (NOM_RATE/10.0)/RQTS_PCT   // Work rates
#define DESIGN_RATE (NOM_RATE/10.0)/DESIGN_PCT
#define CODE_RATE (NOM_RATE/10.0)/CODE_PCT
#define TEST_RATE (NOM_RATE/10.0)/TEST_PCT

// Error creation rates  (Errors/KDSL)
#define RQTS_ERR_RATE 9.25
#define DSGN_ERR_RATE 26.0
#define CODE_ERR_RATE 15.0

// Errors and error detection controls
#define ERR_DET_ADJ 0.7
#define RQTS_ERR_MULT (ERR_DET_ADJ*INITSIZE)
#define DSGN_ERR_MULT (ERR_DET_ADJ*INITSIZE)
#define CODE_ERR_MULT (ERR_DET_ADJ*INITSIZE)
#define REWORK_MULT (.005)   // 5 lines per error for rework
#define PCT_ERR_DET_DSGN (0.50)     // Percent of errors detected in
different phases
#define PCT_ERR_DET_CODE (0.70)
#define PCT_ERR_DET_TEST (0.90)

// Second people pool - compiled in if MORE_PEOPLE is defined
#define MORE_PEOPLE 7.0
#define PROGRAMMERS 101

// Prototype model creation - can be turned on and off
#define PROTO_PCT 15.0
#define PROTO_RATE (NOM_RATE/10.0)/PROTO_PCT
#define PROTO_ERR_MULT RQTS_ERR_MULT
#define PROTO_ERR_RATE 9.0          // Almost same as rqts rate
#define P_RQTS_ERR_RATE (RQTS_ERR_RATE/2.0)
#define P_DSGN_ERR_RATE (DSGN_ERR_RATE/2.0)
#define P_RQTS_RATE (NOM_RATE/10.0)/7.0   // 7% of project
#define P_DSGN_RATE (NOM_RATE/10.0)/13.0  // 13% of project
//   #define PROTO 6                // Define this to turn prototype on

// Resource type identifiers
#define PEOPLE 100


// Actually build the sample model!
```

```
void SampleModel::InitModel()
 {
 Task *rqts, *dsgn, *code, *test, *done, *proto;
 People *r;
 Model *m;
 GraphWindow *g;

 // Create an empty model
 ModelViewer::model = new Model(ModelViewer::Name, SIM_TIME);
 m=getModel();

 // Set Schedule
 m->initSchedule(INIT_SCHEDULE);

#ifdef PROTO
 // Include a prototype phase
 proto = new Task(m, "Prototype", PROTO, INITSIZE, REWORK_MULT);
 rqts = new Task(m,"Requirements", REQUIREMENTS, 0.C,
   RLWORK_MULT, PCT_ERR_DET_DSGN);
 dsgn = new Task(m,"Design", DESIGN, 0.0,REWORK_MULT, PCT_ERR_DET_DSGN);

#else
 // Normal run - no prototype
 rqts = new Task(m,"Requirements", REQUIREMENTS, INITSIZE,
   REWORK_MULT);
 dsgn = new Task(m,"Design", DESIGN, 0.0,REWORK_MULT, PCT_ERR_DET_DSGN);
#endif
 code = new Task(m,"Coding", CODING, 0.0,REWORK_MULT, PCT_ERR_DET_CODE);
 test = new Task(m,"Testing", TESTING, 0.0,REWORK_MULT,
PCT_ERR_DET_TEST);
 done = new Task(m,"Work Done!", DONE, 0.0,REWORK_MULT);

 // Chain the tasks together
#ifdef PROTO
 proto->add(*rqts);
#endif
 rqts->add(*dsgn);
 dsgn->add(*code);
 code->add(*test);
 test->add(*done);

 g = new GraphWindow(this, "Work Done");
 m->addGraph(*g);
#ifdef PROTO
 g->AddGraph(proto, "Prototype");
#endif
 g->AddGraph(rqts, "Requirements");
 g->AddGraph(dsgn, "Design");
 g->AddGraph(code, "Coding");
 g->AddGraph(test,"Testing");
```

```
 // Graph of mandays ert and remaining
 g = new GraphWindow(this, "Mandays");
 m->addGraph(*g);
 g->AddGraph(&(m->getMandays()), "Mandays Expended");
 g->AddGraph(&(m->getEstResDays()), "Estimated Mandays");
 g->AddGraph(&(m->getResDaysRemain()), "Mandays Remaining")

 // Graph Errors
 g = new GraphWindow(this, "Errors Graph");
 m->addGraph(*g);
#ifdef PROTO
 g->AddGraph(proto->getErrors(), "Prototype Errs");
#endif
 g->AddGraph(rqts->getErrors(), "Requirements Errs");
 g->AddGraph(dsgn->getErrors(), "Design Errs");
 g->AddGraph(code->getErrors(), "Code Errs");

 // Create a resource - in this case people
 r=new People(m,"People", PEOPLE, INITPEOPLE, WILL_USE_OT);

 // Add productivities to our resource for each task type
#ifdef PROTO
 r->add(* (new Prod(PROTO, PROTO_RATE, PROTO_ERR_RATE, PROTO_ERR_MULT))
);
 r->add(* (new Prod(REQUIREMENTS,
P_RQTS_RATE,P_RQTS_ERR_RATE,RQTS_ERR_MULT)) );
 r->add(* (new Prod(DESIGN, P_DSGN_RATE, P_DSGN_ERR_RATE,
DSGN_ERR_MULT)) );
#else
 r->add(* (new Prod(REQUIREMENTS,
RQTS_RATE,RQTS_ERR_RATE,RQTS_ERR_MULT)) );
 r->add(* (new Prod(DESIGN, DESIGN_RATE, DSGN_ERR_RATE, DSGN_ERR_MULT))
);
#endif
 r->add(* (new Prod(CODING, CODE_RATE, CODE_ERR_RATE, CODE_ERR_MULT)) );
 r->add(* (new Prod(TESTING, TEST_RATE)) );

// Conditional code section for the case where we want multiple
//  pools of people!
#ifdef MORE_PEOPLE
 r= new People(m, "Programmers", PROGRAMMERS, MORE_PEOPLE, WILL_USE_OT);
 // Make new pool more productive in coding but less productive
 //   in design
 r->add(* (new Prod(REQUIREMENTS,
RQTS_RATE,RQTS_ERR_RATE,RQTS_ERR_MULT)) );
 r->add(* (new Prod(DESIGN, DESIGN_RATE*.5, DSGN_ERR_RATE,
DSGN_ERR_MULT)) );
 r->add(* (new Prod(CODING, CODE_RATE*2.0, CODE_ERR_RATE,
CODE_ERR_MULT)) );
 r->add(* (new Prod(TESTING, TEST_RATE)) );
```

```
#endif

 // Add a graph for overtime
 g = new GraphWindow(this, "Overtime");
 m->addGraph(*g);
 g->AddGraph(&(r->getHoursPerWeek()), "Hours Per Week");

 g = new GraphWindow(this, "Burnout");
 m->addGraph(*g);
 g->AddGraph(&(r->getBurnout()), "Burnout");

 // Initialize the default model step size to 20 days
 setSteps(20);
 };


// This is the main program - taken directly from the
//   ObjectWindows reference - just create an application
//   and tell it to run.
int PASCAL
 WinMain(HANDLE hInst, HANDLE hPrev, LPSTR line, int show)
  {
  ModelApp mapp(hInst,hPrev, line,show);

  mapp.Run();

  return mapp.Status;
  };
```

MODEL.CPP

```
// Object Oriented Software Process Model
// Thesis project by Bradley Smith

#include <spm.h>

// Global seed for NamedBase class - used to generate
//  a unique identifier for each task and resource
IDTYPE NamedBase::Id_seed;

// Hashing function that must be defined for Objects
hashValueType
Value::hashValue() const
{
 return((hashValueType) value);
}

// Function that allows direct value comparisons
int
Value::isLessThan(const Object& testobj) const
 {
 return(value < ((Value &)testobj).getValue());
 }

#ifdef WINDOWS
// Windows function to paint a Value object's number on
//   a window at location specified in the rectangle prect.
void
Value::paintTo(HDC hdc, PRECT prect)
 {
   char s[25];
   sprintf(s,"%.2f", value);
   DrawText(hdc, s, lstrlen(s), prect,
      DT_RIGHT | DT_SINGLELINE);
 }
#endif

// Function that returns the Prod object for a given
//  task_id.  Used to calculate productivity for that task.
Prod &Resource::getProd(IDTYPE task_id)
 {
 ArrayIterator i(*Prods);

 for(i.restart(); i.current() != NOOBJECT; i++)
   if( ((Prod &) i.current()).getType() == task_id)
      return((Prod &) i.current());
 return ((Prod &) NOOBJECT);
 }

// Constructor for resource class
```

```
Resource::Resource(Model *m,LPSTR n,IDTYPE type, double num) :
   NamedBase(n, type, num)
{
model = m;
Prods = new Array(5,0,1);
Prods->ownsElements(TRUE);
m->addResource(*this);
}



// Function to step all values associated with a resource
void Resource::step()
 {
  ArrayIterator i(*Prods);
  NamedBase::step();
  for(i.restart(); i.current() != NOOBJECT; i++)
   ((Prod &) i.current()).step();
 }

// Function to calculate all values for a resource object
void Resource::calculate()
 {
   ArrayIterator i(*Prods);
   NamedBase::calculate();
   for(i.restart(); i.current() != NOOBJECT; i++)
      ((Prod &) i.current()).calculate();
 }

// Function to reset all values for a resource to their initial
//  values.
void Resource::restart()
 {
   ArrayIterator i(*Prods);
   NamedBase::restart();
   for(i.restart(); i.current() != NOOBJECT; i++)
      ((Prod &) i.current()).restart();
 }

// Constructor for the Task class
Task::Task(Model *mod,LPSTR name, IDTYPE type, double v,
      double err_mult, double pct_det) : // Constructor
      NamedBase(name, type, v)
{
model = mod;
NextTasks = new Array(4,0,1);
ConvRate.initValue(1.0);
ErrReworkMult.initValue(err_mult);
PctErrsDet.initValue(pct_det);
WorkDone.initValue(0.0);
model->addTask(*this);
```

```
};

// Member function to calculate the work done and errors generated for
//  a given task, passing work done to all NextTask tasks.
void Task::calculateTask(Resource &r, double num, Prod &p)
 {
 double errs_det, work_passed, work_done;

 // Calculate data members
 ConvRate.calculate();
 Errs.calculate();

 // Calculate the work done in this time step
 work_done = MIN(getValue()*num/getTotRes()
      ,p.getValue() * num * r.getProdMult());
 // Add to error pool
 Errs.addFlow(p.getErrorRate() * work_done *
r.getErrMult()/p.getRigor());
 // Do the work
 addFlow(- work_done);
 WorkDone.addFlow(work_done);

 // Error feedback calculation
 errs_det=(num/getTotRes())*(Errs.getValue()*(getErrorsDetected()-
WorkDone.getValue()*getPctErrsDet()))/p.getErrorDetMult();
 errs_det = MIN(Errs.getValue()*num/getTotRes(), errs_det);
 Errs.addFlow(-errs_det);
 addFlow(errs_det*ErrReworkMult.getValue());

 // Add to other tasks based on conversion rate
 work_passed = work_done * getConvRate();

 // Pass work done to all tasks in the NextTasks list
 ArrayIterator i(*getNextTasks());
 for(i.restart(); i.current() != NOOBJECT; i++)
   ((Task &) i.current()).addWork(work_passed);

 }

 // Compute errors detected in subsequent phases
double
Task::getErrorsDetected()
{
 double d = WorkDone.getValue()*getPctErrsDet();
 ArrayIterator i(*getNextTasks());
 for(i.restart(); i.current() != NOOBJECT; i++)
   d+=((Task &)i.current()).getErrorsDetected() / getConvRate();
 return(d);
}

// Estimate how long it will take to do this
```

```
double
Task::EstimateResDays(double in)
{
 double est;      // Estimate
 ArrayIterator next(*NextTasks);
 if(getResDays() == 0.0 || getValue() == 0.0)
   est = 0.0;
 else
   est = in*getResDays()/getValue(); // Current work rate
 for(next.restart(); next.current() != NOOBJECT; next++)
   est += ((Task &)next.current()).EstimateResDays(in);
 return(est);
}


// A function to step the model n steps forward in time.
void Model::stepModel(int n)
 {
 int i;

 for(i=0; i<n; i++)
   {
   calculate();
   step();
   }
 };

// A function that steps the entire model forward one day
//   in time.
void Model::step()
 {
  if(CurTime+1 > stop)
   return;
  CurTime ++;
  ArrayIterator i(*tasks);
  for(i.restart();i.current() != NOOBJECT; i++)
   ((Task &) i.current()).step();
  ArrayIterator j(*resources);
  for(j.restart(); j.current() != NOOBJECT; j++)
   ((Resource &) j.current()).step();
  ArrayIterator k(*allocations);
  for(k.restart(); k.current() != NOOBJECT; k++)
   ((Allocation &) k.current()).step();
  Mandays.step();
  EstResDays.step();
  ResDaysRemain.step();
  Schedule.step();
  TotalPeople.step();
#ifdef WINDOWS
  ArrayIterator l(*graphs);
  for(l.restart(); l.current() != NOOBJECT; l++)
```

```
   ((GraphWindow &)l.current()).step();
// InvalidateRect(pModelViewer->control->HWindow,NULL, TRUE);
#endif

  }

 // Restart the model to it's initial values
 void Model::restart()
 {
  CurTime = 0.0;
  ArrayIterator i(*tasks);
  for(i.restart();i.current() != NOOBJECT; i++)
   ((Task &) i.current()).restart();
  TotalPeople.restart();
  ArrayIterator j(*resources);
  for(j.restart(); j.current() != NOOBJECT; j++)
   ((Resource &) j.current()).restart();
  Mandays.restart();
  EstResDays.restart();
  flushAllocations();
  Allocate();
#ifdef WINDOWS
  ArrayIterator l(*graphs);
  for(l.restart(); l.current() != NOOBJECT; l++)
   ((GraphWindow &)l.current()).restart();
#endif
  }

// This allocation computes average productivity and then assigns
// resources based on total resource days required.
void Model::Allocate()
 {
 double tot_res_days = 0.0;
 ArrayIterator r(*resources);
 int n;
 Task *task;
 Resource *res;
 Allocation *al;
 flushAllocations();

 // Total the tasks in mandays
 ArrayIterator t(*tasks);
 for(t.restart(); t.current() != NOOBJECT; t++)
   {
    task = &((Task &) t.current());
   task->setResDays(0.0);      // Compute new average work rate
   task->setTotRes(1e-10);     // Reset total resources for this task
   n=0;
    // Find average work rate for this task!
   for(r.restart(); r.current() != NOOBJECT; r++)
      {
```

```
        Prod &p=((Resource &)r.current()).getProd(task->getType());
        if(p != NOOBJECT)
              {
              task->setResDays(task->getResDays()+
                    p.getValue()*((Resource &)r.current()).getProdMult());
              n++;
              }
         }
    // Compute the average
    if(n!= 0)
       {
       // Divide work by average work rate to estimate resource days
       task->setResDays(task->getValue()/
             (task->getResDays() / (double) n));
       }
    else
       task->setResDays(0.0);  // Otherwise no work can be done
    }

  // Now make allocations bases on total resources available
  for(r.restart(); r.current() != NOOBJECT; r++)
    {
    ArrayIterator pd(*((Resource &)r.current()).getProds());
    tot_res_days = 0.0;
    for(pd.restart(); pd.current() != NOOBJECT; pd++)
       {
       tot_res_days += getTask(((Prod
&)pd.current()).getType()).getResDays();
       }
    // Make sure there is work to be done!
     if(tot_res_days != 0.0)
     // Actually make allocations
       for(pd.restart(); pd.current() != NOOBJECT; pd++)
          {
          task = &(getTask(((Prod &) pd.current()).getType()));
            res = &((Resource &) r.current());
          al = new Allocation(task, res, res->getValue()*
                   task->getResDays()/tot_res_days, &((Prod
&)pd.current()));
          task->addTotRes(res->getValue()*task->getResDays()
                   /tot_res_days);   // Add to task total resources
          allocations->add(*al);
          }
    }

 // Compute estimate of resource days required for project!
 EstimateResDays();
 }

void
Model::EstimateResDays()
```

```
 {
 double est = 0.0;
 ArrayIterator t(*tasks);
 for(t.restart(); t.current() != NOOBJECT; t++)
   // Add each tasks estimate to total
   est +=((Task &) t.current()).EstimateResDays(
       ((Task &)t.current()).getValue());
 // Revise our estimate each time step!
 EstResDays.addFlow(est-EstResDays.getValue());

 // Set resource days remaining for each step
 if(Schedule.getValue() > (double) CurTime)
   setResDaysRemain(TotalPeople.getValue()
       *(Schedule.getValue()-(double) CurTime));
 else
   setResDaysRemain(0.0);     // No days remaining in schedule
 };

// Estimate resource shortfall based on mandays remaining
//   and mandays required to finish
double
Model::getResourceShortfall()
{
// Is project under no pressure?
if(ResDaysRemain.getValue() > EstResDays.getValue())
   return 0.0;     // No schedule pressure

// Has schedule already expired?
if(Schedule.getValue() <= (double)CurTime+1.0)
   return EstResDays.getValue();

// Calculate resources needed to complete project
return((EstResDays.getValue()-ResDaysRemain.getValue())
                  / (Schedule.getValue()-(double)CurTime));
}


// Looks up the task with type == id
Task &Model::getTask(IDTYPE id)
  {
  ArrayIterator i(*tasks);
  for(i.restart(); i.current() != NOOBJECT; i++)
   if(id == ((Task &)i.current()).getType())
      return((Task &)i.current());
   return (Task &)NOOBJECT;
  }

// This function recalculates the entire model for the next time step
void Model::calculate()
 {
 // Allocate resources
```

```
 Allocate();
 // Walk allocations and step through tasks
 ArrayIterator a(*allocations);
 for(a.restart(); a.current() != NOOBJECT; a++)
   ((Allocation &)a.current()).calculateTask();

 // Calculate resources and mandays expended
 ArrayIterator r(*resources);
 for(r.restart(); r.current() != NOOBJECT; r++)
   {
   ((Resource &) r.current()).calculate();
    if(EstResDays.getValue() > TotalPeople.getValue())
      Mandays.addFlow(((Resource &)r.current()).getValue());
   }
 TotalPeople.calculate();
 // All done!
 }

// Communication loss table - based on people assigned
double CommLossTable[] =
   {
   0.0,      // 0 people
   .140,     // 10 people
   .245,     // 20 people
   .355,     // 30 people
   .385,     // 40 people
   .400,     // 50 people
   .425,     // 60 people
   .445,     // 70 people
   .460,     // 80 people
   .476,     // 90 people
   .492,     // 100 people
   .505,     // 110 people
   .515,     // 120 people
   .525,     // 130 people
   .535,     // 140 people
   .547,     // 150 people
   .558,     // 160 people
   .567,     // 170 people
   .590,     // 180 people +
   };


// Looks up communication loss value from CommLossTable based on
//   number of people assigned to the project.
double
People::getCommLoss()
 {
 double t=model->TotalPeople.getValue();
 int i;
```

```
 if(t>= 180.0)
   return(CommLossTable[18]);
 i=(int)(t/10.0);
 // Otherwise interpolate into table!
 return(CommLossTable[i]+
   (CommLossTable[i+1]-CommLossTable[i])*(t-(double)i*10.0)/10.0);
 }

#define BURNOUT_DELAY 50.0
#define RECOVERY_DELAY 40.0
// Calculates People resources including burnout effects
void
People::calculate()
 {
 double ot, burn;
 Resource::calculate();
 HoursPerWeek.calculate();


 // Calculate new burnout level based on HoursPerWeek
 ot=HoursPerWeek.getValue()-45.0;
 burn=Burnout.getValue();
 if(ot > 0.0)
   Burnout.addFlow(MIN(1.0-burn, ot/40.0/BURNOUT_DELAY));
 else
   Burnout.addFlow(-burn/RECOVERY_DELAY);
 Burnout.calculate();

 // We can adjust the overtime level based on schedule
 //  pressure if the Auto Overtime feature is enabled
 if(WillUseAutoOT > 0.0)
   setHoursPerWeek(MIN(60.0, 40.0*(1+WillUseAutoOT*
      model->getResourceShortfall()/model->TotalPeople.getValue()) ) );
        // To max of 60 hours per week

 // Calculate Losses to Workforce
 if(getRetentionTime() < 1.0e6)     // Is there any retention time?
   addFlow(-getValue() / getRetentionTime());

 // Calculate increases in training level
 if(getTrainLevel() < 1.0  && getTrainDelay() < 1.0e6)
   TrainLevel.addFlow((1.0-getTrainLevel())/getTrainDelay());

 };

 // Get the overtime multiplier for this group of People
double
People::getOTMult()
 {
 double ot,burn;
 ot=HoursPerWeek.getValue()/40.0;
```

```
 burn = Burnout.getValue();
 // Overtime mult is 1-0.5 * burnout squared
 ot = ot*(1.0-0.5*burn*burn);
 return ot;
 }

#ifdef WINDOWS
void
Model::addGraph(GraphWindow &g)
 {
 graphs->add(g);
 }
#endif
```

MODWIN.CPP

```
// Microsoft Windows Interface -
// Thesis software process model - Bradley Smith

#include <spm.h>

// Global height and width of standard font character
int h_txt, w_txt;

// Function to grab the standard font size for a window
void GetTextSize(HDC hdc, int &w, int &h)
 {
 TEXTMETRIC tm;
 GetTextMetrics(hdc, &tm);
 w=(tm.tmAveCharWidth * 5 )/ 4;
 h=tm.tmHeight + tm.tmExternalLeading;
 }

// ModelViewer constructor - creates a Frame Window
ModelViewer::ModelViewer(LPSTR name)
   :TMDIFrame(name, "menu")
 {
 nsteps = 1;
 ChildMenuPos = 3;
 Name = name;
 }

// The SetupWindow function creates all of the model sub-windows
void
ModelViewer::SetupWindow()
 {
 HDC hdc;
 Array *g;
 // Call inherited SetupWindow() function
 TMDIFrame::SetupWindow();

 // Calculate the text size
 hdc = GetDC(HWindow);
 GetTextSize(hdc,w_txt, h_txt);
 ReleaseDC(HWindow, hdc);

 // Creat control, resource, task and allocation windows
 control= new ControlWindow(this);
 res = new ResourceWindow(this);
 tasks = new TaskWindow(this);
 allocations = new AllocationWindow(this);
 summary = new SummaryWindow(this);
 GetModule()->MakeWindow(control);
 GetModule()->MakeWindow(res);
 GetModule()->MakeWindow(tasks);
```

```
 GetModule()->MakeWindow(allocations);
 GetModule()->MakeWindow(summary);

 // g=getModel()->graphs;
 // ArrayIterator i(*g);
 // for(i.restart(); i.current() != NOOBJECT; i++)
 //    GetModule()->MakeWindow((GraphWindow *) &i.current());
 };

// CMStep() menu function steps entire model forward nsteps.
void ModelViewer::CMStep(RTMessage msg)
 {
 HCURSOR hCursor;
 hCursor=SetCursor(LoadCursor(NULL, IDC_WAIT));
 model->stepModel(nsteps);
 SetCursor(hCursor);
 Refresh();
 };

// CMRestart() menu function restarts entire model
void ModelViewer::CMRestart(RTMessage msg)
 {
  model->restart();
 Refresh();
 };

// Step the model continuously
void
ModelViewer::CMContinue(RTMessage msg)
{
HCURSOR hCursor;
hCursor=SetCursor(LoadCursor(NULL, IDC_WAIT));
model->stepModel(model->stop-model->CurTime);
SetCursor(hCursor);
Refresh();
};

// Refresh() function forces all model windows to redraw
void ModelViewer::Refresh()
 {
 InvalidateRect(res->HWindow, NULL, TRUE);
 InvalidateRect(tasks->HWindow, NULL, TRUE);
 InvalidateRect(allocations->HWindow, NULL, TRUE);
 InvalidateRect(control->HWindow, NULL, TRUE);
 InvalidateRect(summary->HWindow, NULL, TRUE);
 };

// ControlWindow Constructor - creates buttons for user to press
//  on the control window.
ControlWindow::ControlWindow(ModelViewer *m)
   : TWindow(m,"Control")
```

```
{
 int w,h;
 HDC hdc;

 // Calculate default font size
 hdc = GetDC(m->HWindow);
 GetTextSize(hdc, w, h);
 ReleaseDC(m->HWindow, hdc);
 Attr.H = h*4;
 Attr.W = w*40;
 model = m;

 // Create the step, quit and restart buttons
 Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
 new TButton(this,CM_STEP, "&Step", w, 0, w*8, h*3/2,TRUE);
 new TButton(this, IDOK, "&Quit", w*9, 0, w*8, h*3/2,FALSE);
 new TButton(this, CM_RESTART, "&Restart", w*17, 0,
   w*10, h*3/2,FALSE);
 new TButton(this, CM_CONTINUE, "&Continuous", w*27, 0,
   w*12, h*3/2,FALSE);

};

// This function could be used to set up additonal controls
void ControlWindow::SetupWindow()
 {
 TWindow::SetupWindow();
 }

// CMStep() command is called when user presses Step button
//   to step the model forward nsteps.
void ControlWindow::CMStep(RTMessage msg)
 {
 model->CMStep(msg);
 };

// CMRestart() function is called when the restart button is pressed
void ControlWindow::CMRestart(RTMessage msg)
 {
 model->CMRestart(msg);
 };

void
ControlWindow::CMContinue(RTMessage msg)
 {
 model->CMContinue(msg);
 };

// CMOK function is called when the quit button is pressed
void ControlWindow::CMOK(RTMessage msg)
 {
```

```
 model->CloseWindow();
 };

// Paint in this case displays the mandays expended at
//  the bottom of the control window.  Later versions
//  will probably put mandays expended on its own report
//  or graph.
void ControlWindow::Paint(HDC hdc, PAINTSTRUCT _FAR &ps)
 {
 int w,h;
 char s[25];
 RECT rect;
 GetTextSize(hdc, w,h);
 SetRect(&rect, 0, h*2, 18*w, h*3);
 wsprintf(s,"Day %d",(model->getModel())->getCurTime());
 DrawText(hdc, s, lstrlen(s), &rect, DT_CENTER|DT_SINGLELINE);
 };

// The CMNSteps() function can be used to change the user
//  step size in the model.
void ControlWindow::CMNSteps(RTMessage msg)
 {
 int val, ret;

 if(msg.LP.Hi != EN_CHANGE)
   return;
 val = GetDlgItemInt(HWindow, R_NSTEPS,&ret, FALSE);
 model->setSteps(val);
 };


// Initialize a summary window for key outputs
SummaryWindow::SummaryWindow(ModelViewer *mv)
   :TWindow(mv, "Summary")
 {
 m=mv;
 };

void
SummaryWindow::Paint(HDC hdc, PAINTSTRUCT _FAR &ps)
{
char s[100];
Model *mod;
double sz, errs;
RECT r;
mod = m->getModel();

// Write out current time
TextOut(hdc,0,0,"Time", 4);
wsprintf(s, "%d",mod->CurTime);
TextOut(hdc, 10*w_txt, 0, s, lstrlen(s));
```

```
// Write out mandays Expended
TextOut(hdc, 0, h_txt, "Mandays", 7);
SetRect(&r, w_txt*10, h_txt, w_txt*20, 2*h_txt);
(mod->getMandays()).paintTo(hdc, &r);

// Write out Project Size
sz = 0.0;
ArrayIterator i(*mod->getTasks());
for(i.restart(); i.current() != NOOBJECT; i++)
   sz += ((Task &) i.current()).getValue();
sprintf(s, "%f KDSL",sz);
TextOut(hdc, 0, 2*h_txt, "Size", 4);
TextOut(hdc, 6*w_txt, 2*h_txt, s, lstrlen(s));

// Write out total errors per KSDL
errs= 0.0;
for(i.restart(); i.current() != NOOBJECT; i++)
   errs += (((Task &)i.current()).getErrors())->getValue();
sprintf(s,"%f Errs/KDSL", errs/sz);
TextOut(hdc, 0, 3*h_txt,"Errs", 4);
TextOut(hdc, 6*w_txt, 3*h_txt, s, lstrlen(s));
}

// Construct a new resource window by setting the Resources
//   data member.
ResourceWindow::ResourceWindow(ModelViewer *mv)
   :TWindow(mv, "Resources")
 {
 Model *m;
 m=mv->getModel();
 Resources = m->getResources();
 };

// The Paint function repaints the resource window showing
//   the name and number of each resource
void
ResourceWindow::Paint(HDC hdc, PAINTSTRUCT _FAR &ps)
 {
 Resource *r;
 int x=0,y=0;
 RECT rect;
 ArrayIterator i(*Resources);

 // Walk all resources
 for(i.restart(); i.current() != NOOBJECT; i++)
   {
    x=0;
   r= &((Resource &) i.current());
   // Display the resource name
   TextOut(hdc, x, y, r->getName(), lstrlen(r->getName()));;
```

```
   SetRect(&rect, x+20*w_txt, y, x+40*w_txt, y+h_txt);
    // Display the number of resources
   r->paintTo(hdc, &rect);
    y+= h_txt;
   }
 };

// Create a TaskWindow by setting the Tasks data member
TaskWindow::TaskWindow(ModelViewer *mv)
   :TWindow(mv, "Tasks")
 {
 Model *m;
 m=mv->getModel();
 Tasks = m->getTasks();
 };

// Paint to a TaskWindow - display the task names,
//  their work waiting and the errors for each
void
TaskWindow::Paint(HDC hdc, PAINTSTRUCT _FAR &ps)
 {
 Task *t;
 int x=0,y=0;
 RECT rect;
 ArrayIterator i(*Tasks);

 // Walk all tasks
 for(i.restart(); i.current() != NOOBJECT; i++)
   {
    x=0;
   t= &((Task &) i.current());
   // Display the task name
   TextOut(hdc, x, y, t->getName(), lstrlen(t->getName()));;
   SetRect(&rect, x+20*w_txt, y, x+30*w_txt, y+h_txt);
    // Display the work waiting value
   t->paintTo(hdc, &rect);
   SetRect(&rect, x+30*w_txt, y, x+40*w_txt, y+h_txt);
    // Display the number of errors
   (t->getErrors())->paintTo(hdc, &rect);
    y+= h_txt;
   }
 };

// The AllocationWindow constructor - sets the Allocations
//   data member for display of model allocations
AllocationWindow::AllocationWindow(ModelViewer *mv)
   :TWindow(mv, "Allocation of Resources")
 {
 Model *m;
 m=mv->getModel();
 Allocations = m->getAllocations();
```

```
 };

// AllocationWindow Paint function - Displays the task name,
//  resource name, and number of resources allocated for
//  each allocation.
void
AllocationWindow::Paint(HDC hdc, PAINTSTRUCT _FAR &ps)
 {
 Allocation *a;
 Task *t;
 Resource *r;
 int x=0,y=0;
 RECT rect;
 ArrayIterator i(*Allocations);

 // Walk all resources
 for(i.restart(); i.current() != NOOBJECT; i++)
   {
    x=0;
   a= &((Allocation &) i.current());
   t=a->getTask();
    // Display the task name
   TextOut(hdc, x, y, t->getName(), lstrlen(t->getName()));
   x+=15*w_txt;
   r=a->getResource();
    // Display the resource name
   TextOut(hdc, x, y, r->getName(), lstrlen(r->getName()));;
   SetRect(&rect, x+15*w_txt, y, x+25*w_txt, y+h_txt);
   // Display the number of resources allocated
   a->paintTo(hdc, &rect);
    y+= h_txt;
   }
 };

 // Add a point to the graph's point list
 void
 Graph::AddPoint(double v)
  {
  points->add(*(new GPoint(v)));
  }

 // Step graph - add next point to it
 void
 Graph::step()
  {
  double v;
  if((count++ % point_count) == 0)
   {
   AddPoint((v=value->getValue()));
   minvalue= MIN(v, minvalue);
   maxvalue = MAX(v, maxvalue);
```

```
    }
   }

 GraphWindow::GraphWindow(ModelViewer *m, LPSTR name)
   :TWindow(m, name)
   {
   modelv = m;
   minval=0.0; maxval = 0.01;
   // Create graphs array
   graphs = new Array(5,0,1);
   graphs->ownsElements(TRUE);
   }

void
GraphWindow::AddGraph(Value *v, LPSTR name, int np)
 {
 graphs->add(*(new Graph(v, name, np)));
 }

void
GraphWindow::step()
 {
 Graph *g;
 ArrayIterator i(*graphs);
 for(i.restart(); i.current() != NOOBJECT; i++)
   {
   g= &((Graph &)i.current());
   g->step();
   minval = MIN(minval, g->minvalue);
   maxval = MAX(maxval, g->maxvalue);
   }
 Refresh();
 }

 void
GraphWindow::restart()
 {
 Graph *g;
 ArrayIterator i(*graphs);
 minval = 0.0;
 maxval = 0.01;
 for(i.restart(); i.current() != NOOBJECT; i++)
   {
   g= &((Graph &)i.current());
   g->restart();
   minval = MIN(minval, g->minvalue);
   maxval = MAX(maxval, g->maxvalue);
   }
 Refresh();
 }
```

```
#define TXTWID 18
#define MIN_SIZE 75
void
GraphWindow::Paint(HDC hdc, PAINTSTRUCT &ps)
{
 RECT g;     // Area allocated for graphing
 RECT t;     // Text area
 RECT r;            // Client Window area
 int y,x,c, c2;
 LPSTR name;
 char s[20];
 Graph *graph;
 GPoint *p;

 // Determine extents of the graph area
 GetClientRect(HWindow, &r);

 // Allocate a straight TXTWID for text
 g.top = h_txt;
 g.left = 10*w_txt;
 g.bottom = MAX(MIN_SIZE, r.bottom-2*h_txt);
 g.right = MAX(MIN_SIZE, r.right-g.left-TXTWID*w_txt);

 // Allocate area for legend
 t.top = g.top;
 t.left = g.right+w_txt;
 t.bottom = h_txt * graphs->getItemsInContainer() + g.top;
 t.right = t.left + TXTWID*w_txt;

 // Draw axis
 MoveTo(hdc, g.left,g.top);
 LineTo(hdc, g.left,g.bottom);
 LineTo(hdc, g.right, g.bottom);

 // Draw x numbers
 MoveTo(hdc, g.left,g.bottom);
 LineTo(hdc, g.left,g.bottom+h_txt/2);
 TextOut(hdc, g.left-w_txt/2, g.bottom+h_txt/2, "0",1);
 MoveTo(hdc, x=g.left+(g.right-g.left)/2, g.bottom);
 LineTo(hdc, x, y=g.bottom+h_txt/2);
 wsprintf(s,"%d",modelv->getModel()->stop/2);
 TextOut(hdc, x-2*w_txt, y,s,lstrlen(s));
 MoveTo(hdc, g.right, g.bottom);
 LineTo(hdc, g.right, y=g.bottom+h_txt/2);
 wsprintf(s,"%d", modelv->getModel()->stop);
 TextOut(hdc, g.right-2*w_txt, y, s, lstrlen(s));

 // Draw Y Labels!
 LabelYAxis(hdc,g);

 // Draw Text legend
```

```
  y=t.top;
  SetBkMode(hdc, TRANSPARENT);
  ArrayIterator i(*graphs);
  c=0;
  for(i.restart(); i.current() != NOOBJECT; i++)
    {
    name = ((Graph &) i.current()).getName();
    SelectColor(hdc, c++);
    TextOut(hdc,t.left, y, name, lstrlen(name));
    y+=h_txt;
    }

  // Begin drawing points
  c=0;
  for(i.restart(); i.current() != NOOBJECT; i++)
    {
    graph = (Graph *) &i.current();
    SelectColor(hdc,c++);
    // Draw each set of points
    ArrayIterator j(*graph->points);
     c2=0;
    for(j.restart(); j.current() != NOOBJECT; j++)
       {
       x=g.left+(c2*(float)(g.right-g.left))/modelv->getModel()->stop;
    // Scale time axis
       y=g.bottom-(((GPoint &)j.current()).getValue()-minval)
             *(float)(g.bottom-g.top)/(maxval-minval);
       // Draw line
       if(c2==0)
             MoveTo(hdc, x,y);
       else
             LineTo(hdc, x, y);
         // Jump to next point count on x axis
       c2+= graph->point_count;
       }
    }

// Eventually need to add axis code
}

static HPEN hColorPen;

void
SelectColor(HDC hdc, int i)
 {
 COLORREF rgbColor;
 HPEN hNewPen;

 switch(i%4)
   {
   case 0:
```

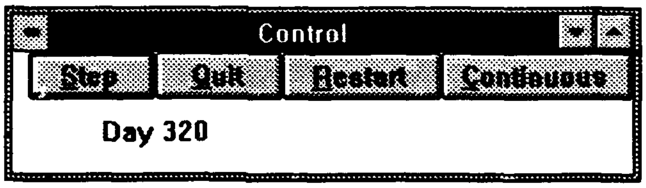
```
      rgbColor = RGB(255,0,0);  // Red
      break;
   case 1:
      rgbColor = RGB(0, 255, 0);    // Green
      break;
   case 2:
      rgbColor = RGB(0,0,255);      // Blue
      break;
   case 3:
      rgbColor = RGB(128,128,128);  // Grey
      break;
    }
 SelectObject(hdc,(hNewPen = CreatePen(PS_SOLID, 1, rgbColor)));
 if(hColorPen)
   DeleteObject(hColorPen);
 hColorPen = hNewPen;
 SetTextColor(hdc, rgbColor);
 }

// Create Y Axis Label
void
GraphWindow::LabelYAxis(HDC hdc, RECT &g)
 {
 double step, start,i;
 int y;
 char s[25];
 // Determine step
 i = (int)log10(maxval-minval);
 step = pow(10.0, (float) i);
 start = ((int)(minval/step)) * step;
 for(; start < maxval; start += step)
   {
   y = g.bottom-((start-minval)*(float)(g.bottom-g.top)/
      (maxval-minval));
   MoveTo(hdc, g.left,y);
   LineTo(hdc, g.left-w_txt/2, y);
   sprintf(s,"%g",start);
   TextOut(hdc, g.left-lstrlen(s)*w_txt-w_txt, y-h_txt/2,
      s, lstrlen(s));
    }
 }
```

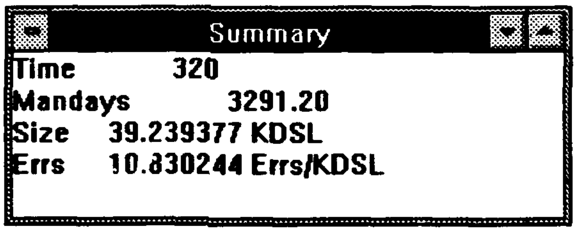
Appendix C

Sample Graphs and Reports

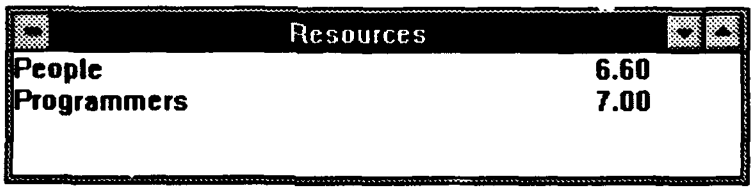
Control Window: Used to step, restart and quit the model.



Summary Window: Displays summary of project statistics including current day, mandays expended, size, and total errors per thousand lines of code.



Resource Window: Displays resource names and number of resources.



Allocation of Resources: Shows the number of resources assigned from each resource object to each task object.

Allocation of Resources

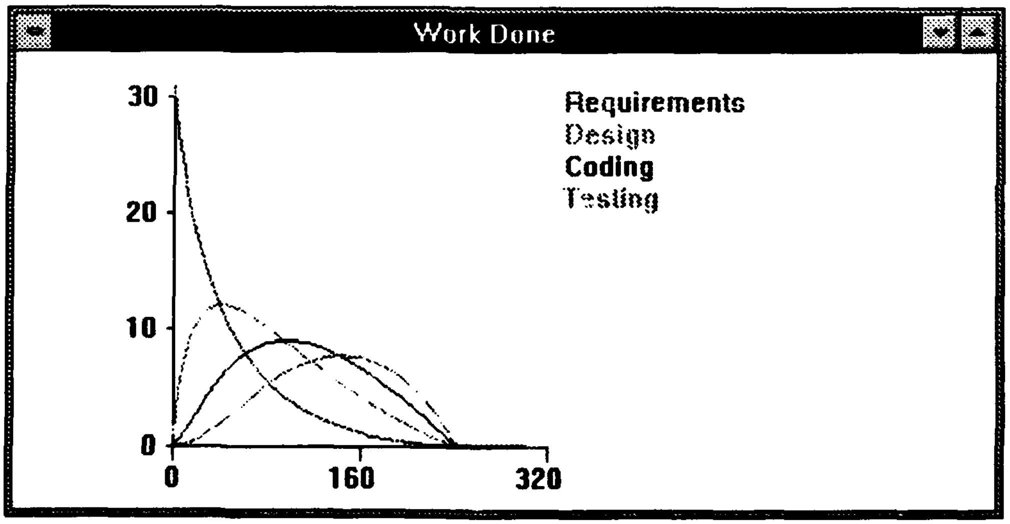
Requirements	People	0.04
Design	People	0.78
Coding	People	2.11
Testing	People	3.67
Requirements	Programmers	0.04
Design	Programmers	0.83
Coding	Programmers	2.23
Testing	Programmers	3.89

Tasks: Report showing the amount of work (in KDSL) for each task and the total number of undetected errors in the task.

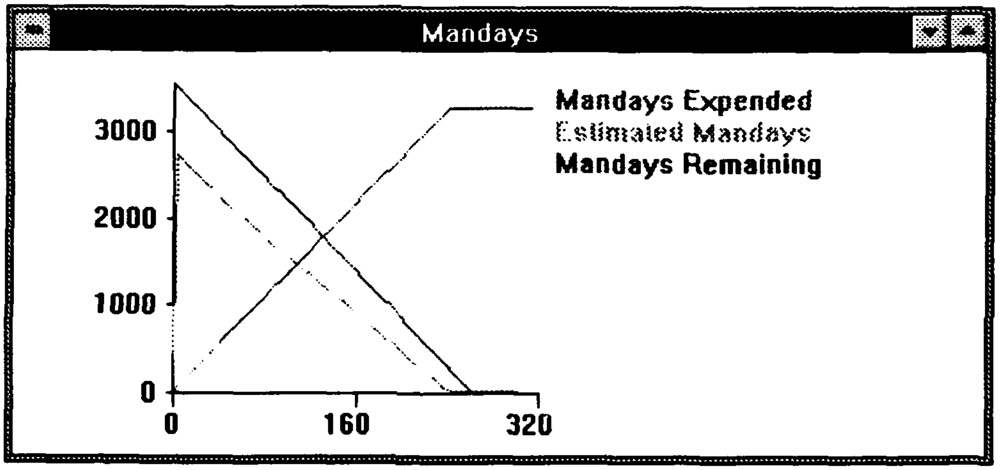
Tasks

Requirements	0.00	17.43
Design	0.00	168.76
Coding	0.00	238.78
Testing	0.00	0.00
Work Done!	39.24	0.00

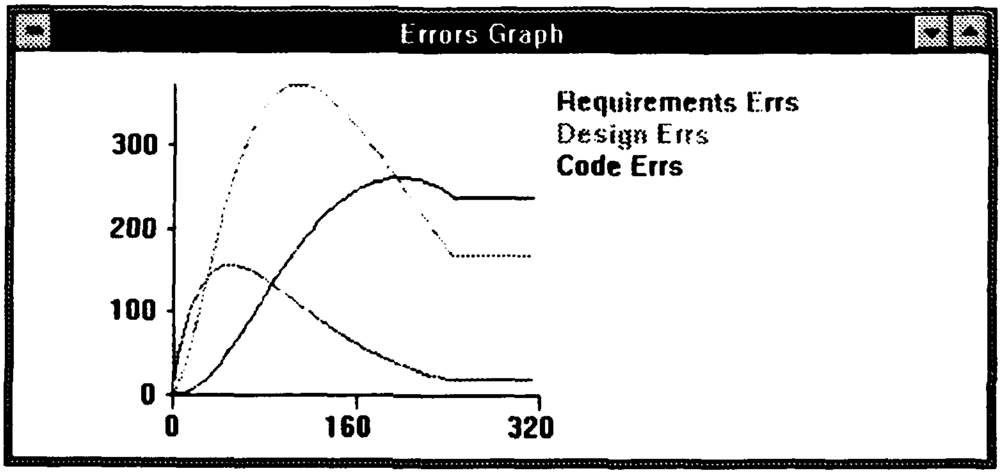
Work Done: Graph showing the work waiting in each task.



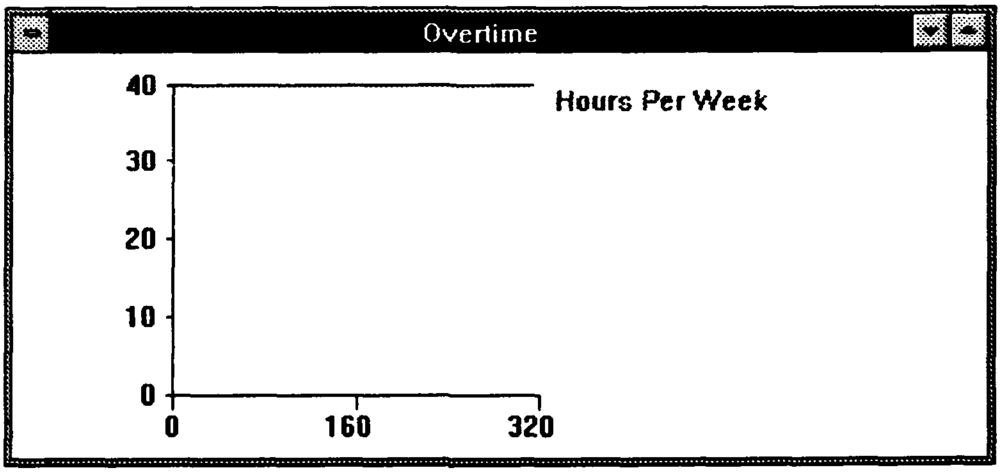
Mandays Graph: Graph showing the number of mandays remaining in the scheduled project versus the number of mandays estimated to complete the project. Total mandays expended to date are also displayed.



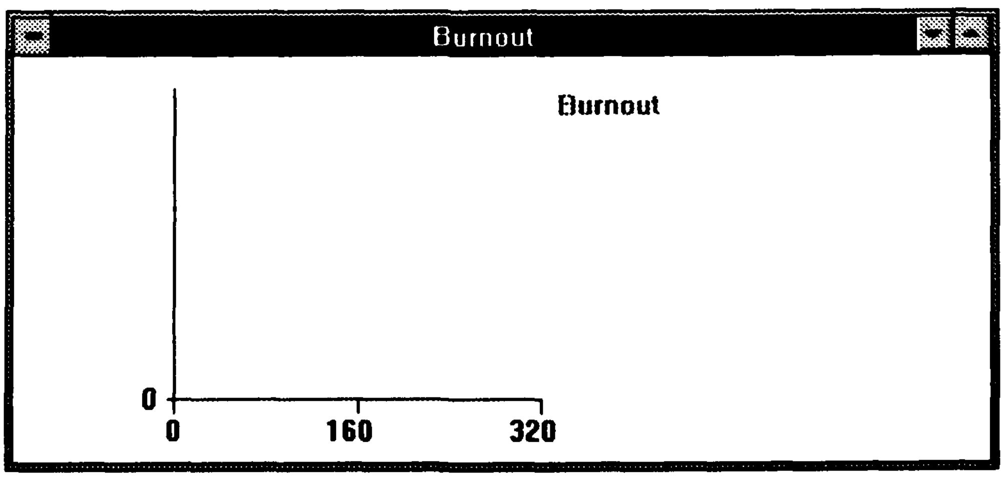
Errors Graph: Displays the total number of undiscovered errors in each task object.



Overtime: Displays a graph of the overtime assigned on the project to date in terms of hours per workweek. Overtime is considered any value over 40 hours per week. In the example shown, no overtime was used.



Burnout: Displays the total burnout level for people assigned to the project. Burnout varies from zero to 100%.



References

1 Tom DeMarco and Timothy Lister, Peopleware: Productive Projects and Teams, Corset House Publishing Co. , New York, 1987, p.188

[2] Barry Boehm, Software Engineering Economics, The COCOMO Software Model, 1980

[3] Roger Pressman, Software Engineering, A Practicioner's Approach, McGraw-Hill, 1992 p. 87

[4] Pressman, p. 83-91

[5] High Performance Systems Inc, Stella II User's Guide, 1990

[6] Tarek Abdel-Hamid and Stuart Madnick, Software Project Dynamics, 1991

[7] B. Smith, A. Clough, R. Vidale, N. Nguyen, S. Ahmed, The Software Process Model, Draper Labs, May 1992

[8] Smith, Clough, Vidale, Ahmed, and Nguyen

[9] Bradley Smith, Proposal for Independent Study of an Object Oriented Software Process Model, May 1992

[10] Smith, Clough, Vidale, Ahmed, and Nguyen

[11] Abdel-Hamid and Madnick

[12] Military Standard 2167A The Software Life Cycle

[13] Grady and Caswell, Software Metrics Prentice Hall, *1987*, p. *24-25*

[14] Grady and Caswell, p. 25

[15] Smith, Clough, Vidale, Ahmed, and Nguyen, p. 9

[16] Smith, Clough, Vidale, Ahmed, and Nguyen, p. 11

[17] Boehm, p. 381-3

[18] Grady and Caswell

[19] Abdel-Hamid and Madnick

[20] Grady and Caswell, p. 224

[21] Abdel-Hamid and Madnick

[22] Abdel-Hamid and Madnick, p. 68

[23] Abdel-Hamid and Madnick, p. 83

[24] Abdel-Hamid and Madnick, p. 87

[25] Smith, Clough, Vidale, Ahmed, and Nguyen, p. 22

[26] Abdel-Hamid and Madnick

[27] Smith, Clough, Vidale, Ahmed, and Nguyen

[28] Borland International, Borland C++ Programmer's Guide, Version 3.1, 1992

[29] Borland International, ObjectWindows for C++ User's Guide, Version 3.1, 1992

[30] Microsoft Inc., Microsoft Windows User's Guide, Version 3.1, 1992

[31] Smith, Clough, Vidale, Ahmed, and Nguyen

[32] Boehm

[33] Smith, Clough, Vidale, Ahmed, and Nguyen

[34] Roger S. Pressman, Software Engineering, Roger S. Pressman, Third Edition, p. 26